Open GIS Consortium, Inc.

OpenGIS® Simple Features Specification

For OLE/COM

Revision 1.1

OpenGIS Project Document 99-050

Release Date: May 18, 1999

WARNING: The Open GIS Consortium (OGC) releases this specification to the public without warranty. It is subject to change without notice. This specification is currently under active revision by the OGC Technical Committee.

Requests for clarification and/or revision can be made by contacting the OGC at revisions@opengis.org.

NOTICE

The information contained in this document is subject to change without notice.

The material in this document details an Open GIS Consortium specification in accordance with the license and notices set forth on this page. This document does not represent a commitment to implement any portion of this specification in any company's products.

WHILE THE INFORMATION IN THIS PUBLICATION IS BELIEVED TO BE ACCURATE, THE OPEN GIS CONSORTIUM AND THE COMPANIES LISTED ABOVE MAKE NO WARRANTY OF ANY KIND WITH REGARD TO THIS MATERIAL INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. The Open GIS Consortium and the companies list above shall not be liable for errors contained herein or for incidental or consequential damages in connection with the furnishing, performance or use of this material.

The copyright holders list above acknowledge that the Open GIS Consortium (acting itself or through its designees) is and shall at all times be the sole entity that may authorize developers, suppliers and sellers of computer software to use certification marks, trademarks, or other special designations to indicate compliance with these materials.

RESTRICTED RIGHTS LEGEND. Use, duplication, or disclosure by government is subject to restrictions as set forth in subdivision (c)(1)(ii) of the Right in Technical Data and Computer Software Clause at DFARS 252.227.7013

OpenGIS® is a trademark or registered trademark of Open GIS Consortium, Inc. in the United States and in other countries.

Table of Contents

0 Preface

0.1 Submitting Companies

The following companies submitted this implementation specification in response to the OGC Request 1, Open Geodata Model Working Group, A Request for Proposals: OpenGIS Features (OpenGIS Project Document Number 96-021):

- Camber Corporation
- Environmental Systems Research Institute (ESRI)
- Intergraph Corporation
- Laser-Scan, Ltd.
- MapInfo Corporation
- Smallworldwide, plc.

0.2 Submission Contact Points

All questions about the joint submission should be directed to:

Ed Runnion
Camber Corporation
635 Discovery Drive
Huntsville, AL 35806-2801
205-922-3590

David Beddoe
ESRI National Accounts
2070 Chain Bridge Road, Suite 180
Vienna, VA 22182
(703)-506-9515

Sam Bacharach
Intergraph Corporation
2051 Mercator Drive
Reston, VA 20191
703-264-5711

David Arctur, Ph.D.
Laser-Scan, Inc.
45635 Willow Pond Plaza
Sterling, VA 20164

John Reilly
MapInfo Corporation
One Global View
Troy, NY 12180
518-285-7229

Peter Batty
Smallworld Systems, Inc.
5600 Greenwood Plaza Blvd.
Englewood, CO 80111
303-779-6980

0.3 Document Conventions

Courier New font is used to identify code segment and names.

0.4 Revision History

Revision 1.0 includes the following changes from Revision 0:

- Replaced the term “ADC” with “RDS” and updated the accompanying text in the Overview that discusses RDS. The source for this change was proposal #1 from Revision Request 97-407.

- Adam Gawne-Cain, Cadcorp Ltd, 8th March 1999: Replaced Geometry and SpatialReference IDL code segments with latest versions. This means that potential implementors can cut/paste the definative IDL from this document Also editted descriptive text to match latest version of these IDL files.

0.5 Editorial Notes

1 Overview

The Open GIS Consortium, Inc. vision statement states that *"OGC envisions the full integration of geospatial data and geoprocessing resources into mainstream computing and the widespread use of interoperable geoprocessing software and geospatial data products throughout the information infrastructure."* (http://www.opengis.org/vision.html) The attached specification is founded in two fundamental principles that are critical for the GIS industry to reach its goal of open geodata and geoprocessing interoperability "throughout the information infrastructure." Those two principles are summed up in the following statement:

- The GIS database, by nature, is a *fundamental component of a much larger corporate information infrastructure*, and must, therefore*, integrate and leverage accepted standards for enterprise-wide information management.*

In other words, the GIS is an extension of the overall corporate information system that today is "standardized" on relational database management systems and Microsoft Windows applications. Going forward however, the database management systems being implemented may be relational, object-relational, or some yet to be determined data storage technology. Therefore, the GIS industry needs to adopt a direction that maximizes the opportunity to leverage today's relational databases while at the same time provides a direction towards emerging data management technologies.

As an OLE/COM based proposal, current Microsoft technologies for database access were evaluated with respect to geographic information processing. These technologies included ODBC, DAO, RDO, ADO and OLE DB. ADO specifically provides the OLE Automation object oriented standards for accessing and manipulating databases, additionally OLE Automation, as a language independent technology, is quickly becoming the standard for application customization and integration. These paradigms match the needs of GIS data access quite well; *GIS can, and should, be considered a database problem with the additional requirements being geodetic coordinate systems, geometry, and graphics display*. This specification addresses these additional problems with GIS specific interfaces above and beyond the current interfaces available through current Microsoft data access technologies.

This specification does *not* take on the responsibility of database technology interfaces however, as many data sources are not *true* databases. This specification is designed to take full advantage of accepted industry standards thus providing a geographic standard and evolution of that standard that minimizes the impact on the GIS technology providers and yet provides the GIS users with the interoperability and integration they demand. This strategy is not unique to geographic information systems, and as such, Microsoft has leveraged its success and experience with DAO and RDO to provide extensible and robust data access technology through **OLE DB**. This specification is, then, based on use of the OLE DB and ADO facilities provided by Microsoft.

> “OLE DB is a set of OLE interfaces that provide applications with uniform access to data stored in diverse information sources, regardless of location or type. These interfaces allow data sources to share their data through common interfaces without having to implement database functionality not native to the datastore.
>
> OLE DB is a freely published specification designed with industry-wide participation through Microsoft's Open process. OLE DB is a developing industry standard for data access to and manipulation of both SQL and non-SQL data sources. This provides consistency and interoperability in an enterprise's network, from the mainframe to the desktop.” (http://www.microsoft.com/oledb)

Going forward, Microsoft has embraced OLE DB as the foundation for data access within the OLE/COM environment. Microsoft states, “OLE DB is the fundamental Component Object Model (COM) building block for storing and retrieving records and unifies Microsoft's strategy for database connectivity. It will be used throughout Microsoft's line of applications and data stores.”

As background, Microsoft defines OLE DB as follows:

> “OLE DB is a specification for a set of data access interfaces designed to enable a multitude of data stores, of all types and sizes, to work seamlessly together. These interfaces comprise an industry standard for data access and manipulation that can ensure consistency and interoperability in a heterogeneous world of data and data types.
>
> OLE DB goes beyond simple data access by partitioning the functionality of a traditional relational database into logical components, and the events needed for those components to communicate. Developers can use these interfaces to define very simple data providers as well as fully relational databases. This is a strategic part of Microsoft's enterprise infrastructure for component-based computing. Components can be thought of as the combination of both process and data into a secure, reusable object. As a result, it often makes sense to treat components as both consumers and providers of data at the same time. Since the OLE DB specification is a definition of how databases interoperate at various levels, components can be built using OLE DB to behave as a table, even though very complex computing processes can actually occur between the data sources and the consuming applications. This capability will have considerable impact on how multi-tier applications are assembled.” (http://www.microsoft.com/oledb/prodinfo/wpapers/wpapers.htm)

Figure 1.1 presents a conceptual view of the solution and architecture presented in this proposal:

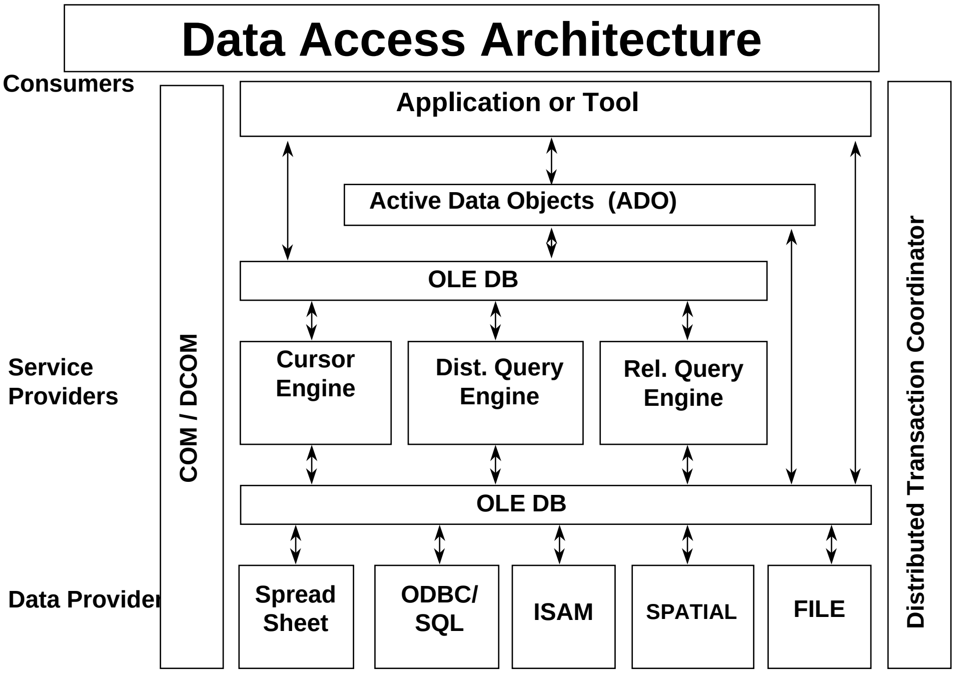


Figure 1.1—Data Access Architecture in the OLE/COM environment

The benefits of an OLE DB based OpenGIS architecture go well beyond the inherent benefits of OLE DB itself. OLE DB is a useful technology in its own right, but becomes much more powerful when leveraging other enterprise technologies provided by Microsoft and other third party developers. For example:

- Distributed Transaction Coordinator—Microsoft's OLE-based transaction product, will utilize OLE DB for data access and makes it possible to coordinate transactions spanning multiple, diverse OLE DB data sources.

- ODBC—OLE DB data consumers, both applications and development tools, have full access to all ODBC drivers through OLE DB and the OLE DB/ODBC Provider (code named Kagera).

- Index Server—Microsoft Index Server works with Windows NT Server 4.0 and Internet Information Server 2.0 to provide your organization access to all of the documents stored on your intranet or Internet site. Index Server is an OLE DB data provider.

- Remote Data Service— The Microsoft® Remote Data Service (RDS) delivers a new Web data access technology that allows developers to create data-centric applications within ActiveX™-enabled browsers such as Microsoft® Internet Explorer. RDS creates a framework that permits easy interaction with ODBC databases on corporate intranets and over the Internet. In addition, the design of the Microsoft® Remote Data Service offers a programming model that leverages the knowledge of Visual Basic® developers. RDS provides the advantages of client-side caching of data results, updateable data, and support for data-aware ActiveX controls. This capability was previous called Active Data Connection (ADC). ADC has now been integrated with ActiveX Data Objects (ADO) to provide data remoting within the same programming model as ADO. This makes it easier to design, code, and deploy both Web-based and LAN-based applications. To clarify the relationship of ADC to ADO, ADC is now known as the Remote Data Service (RDS), a feature of ADO. RDS goes beyond the current generation of Web data access tools by allowing clients to update the data they see (http://www.microsoft.com/data/rds/).

As an enterprise-wide information component the GIS community now has the opportunity to provide industry wide geodata and geoprocessing interoperability while also leveraging mainstream information systems processing resources and technologies.

2 Architecture

The architecture for this specification can be classified into three major components. They are

- **OLE DB** for implementing data providers
- **ADO** for presenting a simplified data access model on top of OLE DB
- **Geometry and Spatial References** for detailed geometry and reference operations

Each of these components is implemented using the Microsoft Component Object Model (COM). ADO and the Geometry and Spatial Reference objects implement IDispatch and are therefore accessible to rapid development languages such as Visual Basic, Java, and Power Builder. All COM interfaces are accessible from lower level languages such as C++ for optimal performance.

2.1 Data Access

2.1.1 OLE DB Overview

In the OLE DB specification, Microsoft writes:

> *OLE DB is a set of OLE interfaces that provide applications with uniform access to data stored in diverse information sources. These interfaces support the amount of DBMS functionality appropriate to the data source, enabling it to share its data.*

As such, OLE DB is an ideal interface for exposing geographic data. One of the principal advantages to exposing and consuming data via the OLE DB interface is that geographic data can then be easily integrated with other databases and office applications. It also means that a wider variety of data can also be utilized within geographic applications such as GIS. It yields a true GIS system by allowing the “G” to be tightly integrated with the corporate “IS”.

There has been no attempt to reproduce the OLE DB specification. The interested reader should consult the Microsoft OLE DB web pages or the OLE DB SDK for details.

The OLE DB architecture diagram shown in Figure 2.1 has been extracted from a Microsoft presentation on OLE DB. It and the preceding table illustrate the interaction between three fundamental categorizations of software:

Category	Relationship to geographic applications
Data Providers	Vendors of commercial GIS software should provide OLE DB data sources which expose their spatial data as well as any attribute data associated with it. Data providers or software vendors may also elect to expose government and other commercially available data sources such as DCW and Tiger.
Service Providers	Geographic services may also be supplied by software vendors or end users. These services might include spatial query processors, buffer zone services, geocoding services, or network analysis services.
Consumers	One of the most significant direct consumers of OLE DB is the ADO interface. The ADO interface consumes OLE DB and then projects a simpler programming model for accessing data.

Table 2.1—Categories of software and their relationship to geographic applications

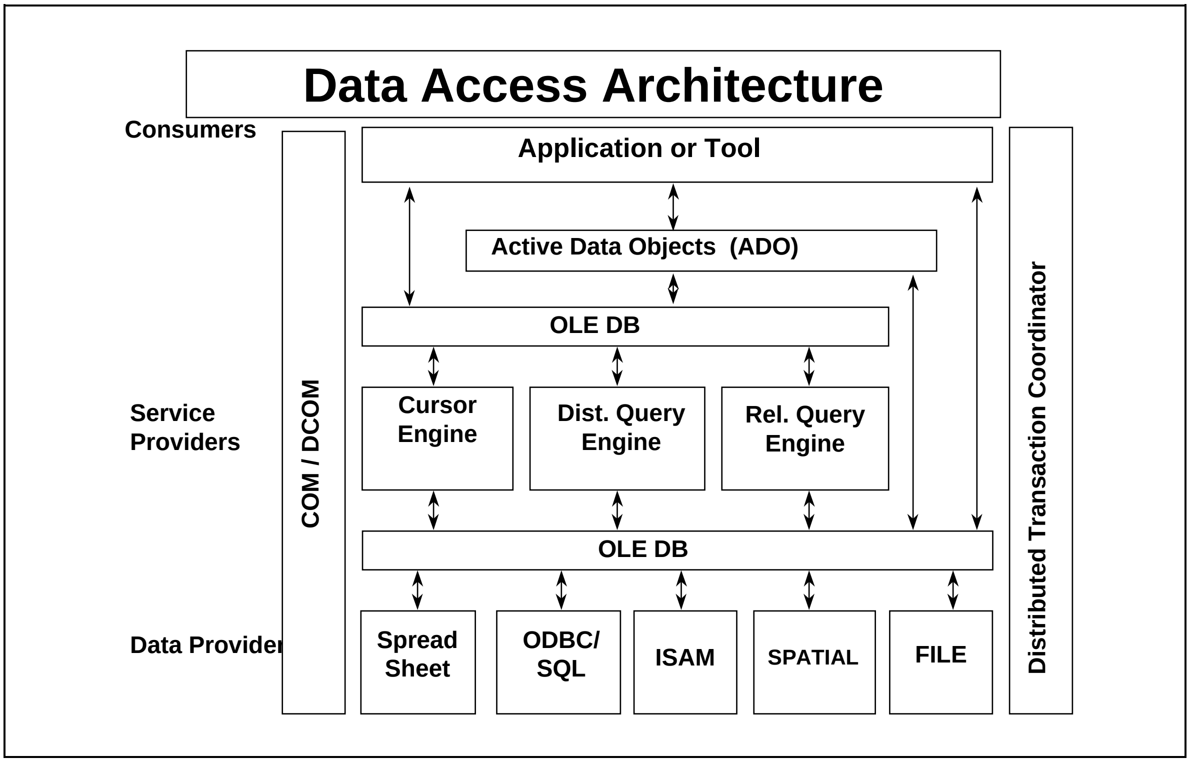


Figure 2.1— Microsoft OLE DB Data Access Architecture

2.1.2 Data Providers

2.1.2.1 Data Provider Overview

The Data Provider category is the most fundamental set of components that must be implemented in order to allow geographic data to be shared among different applications. These applications may be as diverse as data collection, analysis or simple viewing. With OLE DB interfaces to relevant geographic data, customers and other software vendors will be able to view and analyze heterogeneous collections of data from a wide range of data sources without first trying to convert them all to a compatible data format.

Data providers wishing to simply expose data to services and client applications are only required to implement the minimal set of OLE DB interfaces. The minimal set of interfaces for Rowsets is shown in Figure 2.2. Clients must be prepared to work with a data server that implements only the minimal set of interfaces.

Data providers wishing to allow simple creation and modification of data must implement additional OLE DB interfaces (Figure 2.3). The data provider must always be sure that the integrity of the data is preserved according to the rules of valid data for that particular data set. One data provider may allow any geometry to be input which meets the requirements of valid OGIS data. A second data provider may require that geometric data be limited to a smaller set of geometries and be *topologically clean* because of a supporting topological index. This specification does not attempt to define any data validation rules others than those described in the Geometry and Spatial Reference objects.

Data providers with sophisticated relationship models may elect to expose custom COM interfaces which only allow data to be modified using a special set of operations. An example would be a data provider for parcel data (Figure 2.4). The custom interface for this may only expose methods for splitting and merging parcels. The underlying implementation may modify several tables and keep a history of the operation. Although a custom interface is used for modifying the data, the standard OLE DB interfaces should be implemented to expose the data to simpler applications such as viewers. Of course, custom interfaces could be implemented which allow more sophisticated clients to view more of the underlying data structures for analysis purposes. This specification does not attempt to describe the details of any such models or interfaces.

It is therefore clear that the OLE DB data provider interfaces scale well. They are easily extended using COM mechanisms to handle sophisticated data modeling issues.

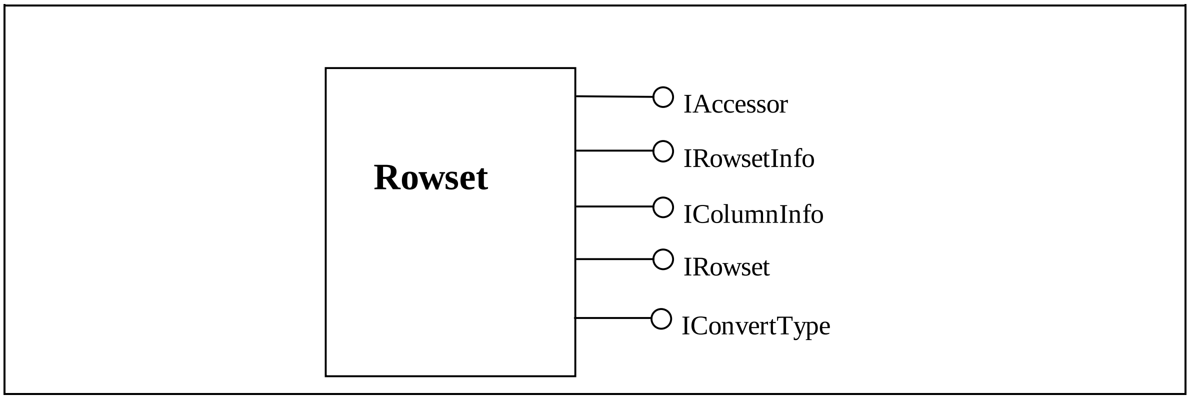


Figure 2.2— Minimum Rowset implementation for a read only data source

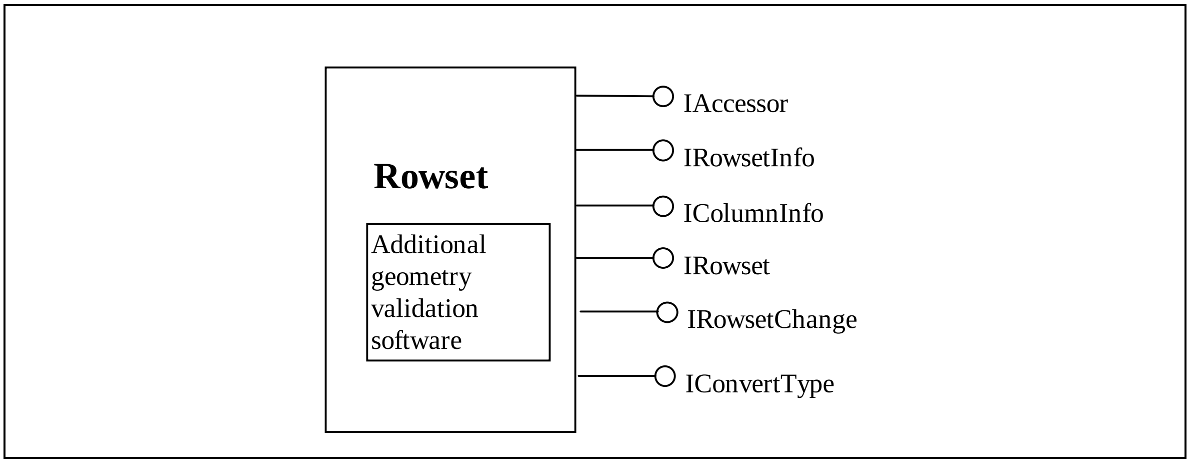


Figure 2.3—Minimum Rowset implementation for a data source that allows modifications

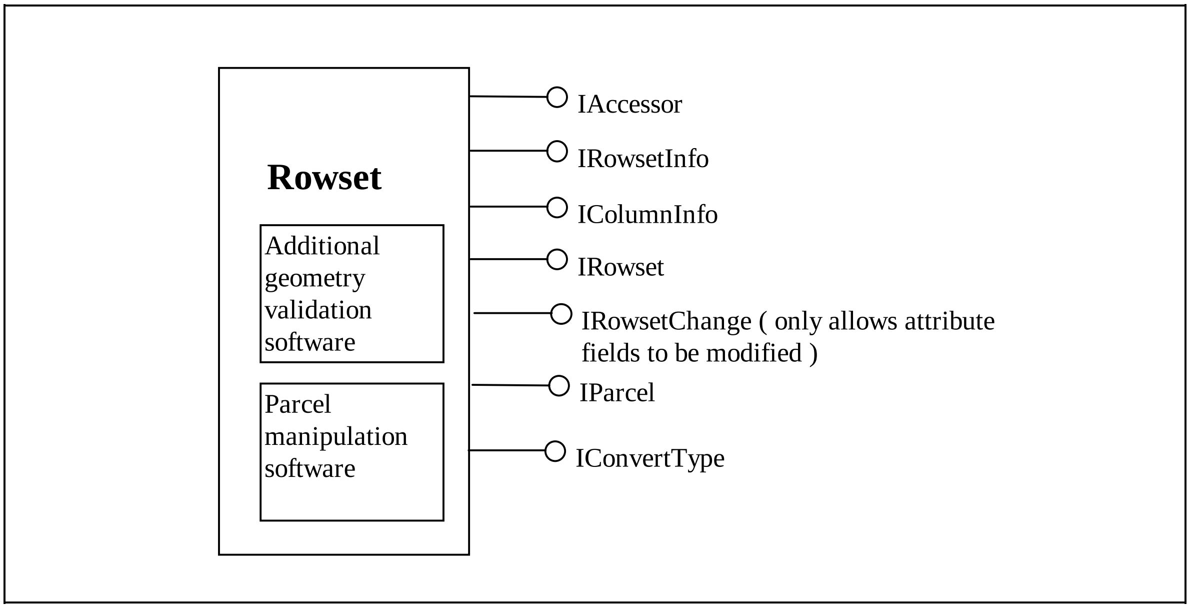

Rowset
Additional geometry validation software
Parcel manipulation software
IAccessor
IRowsetInfo
IColumnInfo
IRowset
IRowsetChange (only allows attribute fields to be modified)
IParcel
IConvertType

Figure 2.4—Custom implementation for sophisticated modeling

2.1.2.2 Requirements for Data Providers

There is no strict “level of compliance” scheme for OGIS data servers. Clients are expected to use standard COM and OLE DB techniques for determining if a data server supports a specified type of functionality. It is the client’s responsibility to decide what action it will take if a data provider fails to support a given service.

The minimum level of support that a data server must pass is:

- Support the minimum set of interfaces as defined by the OLE DB standard
- Require providers to register support for the “OGISDataProvider” component category.
- Geometry values must be exposed as Well Know Binaries (WKB) as described in Section 4.

Data providers which do not support schema rowsets or IColumnsRowset::GetColumnsRowset are encouraged to name their geometry columns “OGIS_GEOMETRY”. Clients are also encouraged to look for columns with this name if the provider does not support the schema methods noted above.

A data provider which wishes to provide additional GIS Metadata and Geometry information to the client, should do so in compliance with Sections 3.1 and 3.2

2.1.3 Data Consumers (ADO)

ADO is a Microsoft implementation of an OLEDB data consumer. While it is not an OLEDB data provider, its purpose is to provide the essence of OLEDB data to automation clients via a standard automation server. This allows easy access to data from high level languages such as Visual Basic and Power Builder. The ADO model is analogous to, but much simpler than the OLEDB model. The fundamental objects in the ADO object hierarchy are:

- Connection—This bundles the DSO and the Session and allows the automation client to connect to the data source, access schema information and begin posing queries.
- Command—This exposes the OLEDB Command capabilities, allowing queries to be posed (Including spatial queries for OGIS data servers).
- Recordset—This is the result of a query or a request for information.
- Field—Column values that one would access via IRowset::GetData in OLEDB are accessed as the Value property of Field objects in the Recordset’s Fields collection.

Note that, via COM, the consumer can use QueryInterface to determine the capabilities of the provider. With ADO, the automation client must be prepared to catch exceptions thrown when the client attempts an activity not supported by the underlying Data Source. For example, a data provider may not support IRowsetLocate (no backwards scrolling), and so ADO’s Recordset.MovePrevious and Recordset.Bookmark may throw exceptions. Clients should be prepared for errors so that they may work with a wide variety of data providers.

Geometry is served up through Well-known Binaries (WKB). This allows a user to then construct a geometry object or use the WKB directly. An example of code that accesses geometry is shown below.

```
Dim geom as IGeometry
Dim rs as ADODB.Recordset
Dim fld as ADODB.Field
Dim factory as IGeometryFactory
Dim srs as ISpatialReference

           ‘ Insert code to initialise factory and srs

           rs.open "Schools”, connectString
           set fld = rs(“OGIS_GEOMETRY”)
           Do Until rs.EOF        ' Do until end of recordset
               set geom = factory.CreateFromWKB( fld.value , srs )
               AddToDisplay geom
               rs.MoveNext'       ' Move to next record.
           Loop
```

2.2 Geometry Object Model

This section describes the object model for geometry. It is Distributed Computing Platform neutral and uses OMT notation. The object model for geometry is shown in Figure 2.5. The base Geometry class has subclasses for Point, Curve, Surface and Geometry Collection. Each geometric object is associated with a Spatial Reference System, which describes the coordinate space in which the geometric object is defined.

Figure 2.5 is based on extending the Geometry Model specified in the OpenGIS Abstract Specification with specialized zero-, one- and two-dimensional collection classes named MultiPoint, MultiLineString and MultiPolygon for modelling geometries corresponding to collections of Points, LineStrings and Polygons respectively. MultiCurve and MultiSurface are introduced as abstract superclasses at this RFP that generalize the collection interfaces to handle Curves and Surfaces. The figure shows aggregation lines between the leaf collection classes and their element classes, the aggregation lines for non-leaf collection classes are described in the text.

The attributes, methods and assertions for each geometry class are described below. In describing methods, *this* is used to refer to the receiver of the method (the object being messaged). The scope of the methods and attributes is based on the scope of RFP1 (SimpleFeatures).

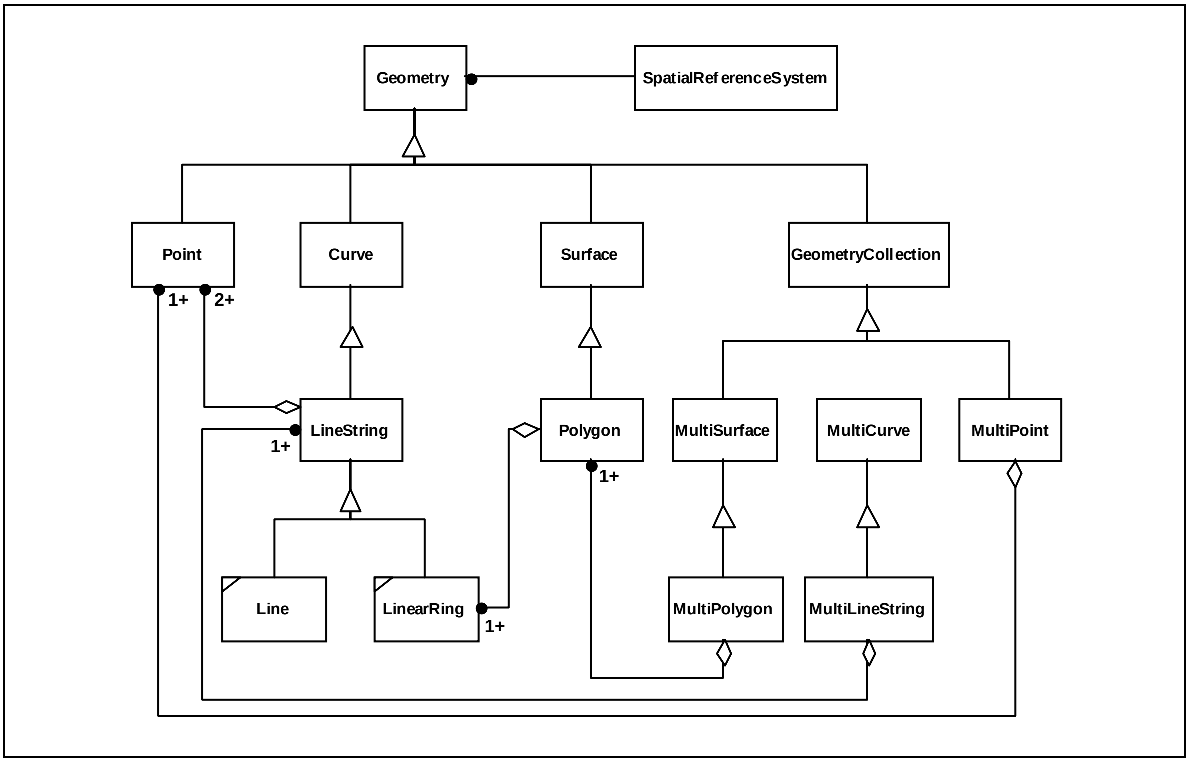


Figure 2.5—Geometry Class Hierarchy

2.2.1 Geometry

Geometry is the root class of the hierarchy. Geometry is an abstract (non-instantiable) class. All geometry objects must support the IGeometry interface.

The instantiable subclasses of Geometry defined in this specification are restricted to 0, 1 and two-dimensional geometric objects that exist in two-dimensional coordinate space ($\Re^2$).

All instantiable geometry classes described in this specification are defined so that valid instances of a geometry class are topologically closed (i.e. all defined geometries include their boundary).

2.2.1.1 Attributes of Geometry

Property **Dimension** As Long—The inherent dimension of *this* geometric object, which must be less than or equal to the coordinate dimension. This specification is restricted to geometries in two-dimensional coordinate space.

Property **IsEmpty** As Boolean—Returns TRUE if *this* geometry is the empty geometry . If true, then *this* geometry represents the empty point set, ∅, for the coordinate space.

Property **IsSimple** As Boolean—Returns TRUE if the geometry has no anomalous geometric points, such as self intersection or self tangency. The description of each instantiable geometric class will include the specific conditions that cause an instance of that class to be classified as not simple.

Property **SpatialReference** As ISpatialReference—Returns the Spatial Reference System for *this* geometry.

2.2.1.2 Basic Methods on Geometry

Function **Clone**() As IGeometry —Return a copy of *this* geometry.

Function **Envelope**() As IGeometry—The minimum bounding box for *this* geometry, returned as a geometry. The polygon is defined by the corner points of the bounding box ((MINX, MINY),(MAXX, MINY), (MAXX, MAXY), (MINX, MAXY), (MINX, MINY)).

Sub **Extent2D**(minX As Double, minY As Double, maxX As Double, maxY As Double) —The minimum bounding box for *this* geometry.

Function **Project**(newSystem As ISpatialReference) As IGeometry– Projects coordinates of *this* geometry into the specified coordinate space, returning a new geometry.

Sub **SetEmpty**()—Sets *this* geometry to the empty geometry, making *this* geometry represents the empty point set, ∅, for the coordinate space.

2.2.1.3 Methods for testing Spatial Relations between geometric objects :

The methods in this section are defined and described in more detail following the description of the sub types of Geometry. These methods are part of the optional ISpatialRelation interface, which geometry objects may support.

Function **Contains**(other As IGeometry) As Boolean— Returns TRUE if this geometry ‘spatially contains’ another geometry.

Function **Crosses**(other As IGeometry) As Boolean— Returns TRUE if this geometry ‘spatially crosses’ another geometry.

Function **Disjoint**(other As IGeometry) As Boolean— Returns TRUE if this geometry is ‘spatially disjoint’ from another geometry.

Function **Equals**(other As IGeometry) As Boolean—Returns TRUE if this geometry is ‘spatially equal’ to another geometry.

Function **Intersects**(other As IGeometry) As Boolean— Returns TRUE if this geometry ‘spatially intersects’ another geometry.

Function **Overlaps**(other As IGeometry) As Boolean— Returns TRUE if this geometry ‘spatially overlaps’ another geometry.

Function **Touches**(other As IGeometry) As Boolean— Returns TRUE if this geometry ‘spatially touches’ another geometry.

Function **Within**(other As IGeometry) As Boolean— Returns TRUE if this geometry is ‘spatially within’ another geometry.

A second optional spatial relationship interface, ISpatialRelation2, may be used for generalised spatial testing with the Relate function.

Function **Relate**(other As IGeometry, intersectionPatternMatrix As String) As Boolean— Returns TRUE if this geometry is spatially related to another Geometry, by testing for intersections between the Interior, Boundary and Exterior of the two geometries as specified by the values in the intersectionPatternMatrix.

2.2.1.4 Methods that support Spatial Analysis

These methods are part of the optional ISpatialOperator interface, which geometry objects may support.

Function **Boundary**() As IGeometry)—Returns the closure of the combinatorial boundary of the geometry.. Because the result of this function is a closure, and hence topologically closed, the resulting boundary can be represented using representational geometry primitives.

Function **Buffer**(Distance As Double) As IGeometry—Returns a geometry that represents all points whose distance from this geometry is less than or equal to distance. Calculations are in the Spatial Reference System of this geometry.

Function **ConvexHull**() As IGeometry—Returns a geometry that represents the convex hull of this geometry.

Sub **Difference**(other As IGeometry, result As IGeometry) —Returns a geometry that represents the point set difference of the source geometry with anotherGeometry.

Function **Distance**(other As IGeometry) As Double—Returns the shortest distance between any two points in the two geometries as calculated in the spatial reference system of *this* geometry.

Sub **Intersection**(other As IGeometry, result As IGeometry) —Returns a geometry that represents the point set intersection of the source geometry with anotherGeometry.

Sub **SymmetricDifference**(other As IGeometry, result As IGeometry) —Returns a geometry that represents the point set symmetric difference of the source geometry with anotherGeometry.

Sub **Union**(other As IGeometry, result As IGeometry) —Returns a geometry that represents the point set union of the source geometry with anotherGeometry.

2.2.2 Geometry Collection

A GeometryCollection is a geometry that is a collection of 1 or more geometries.

All the elements in a GeometryCollection must be in the same Spatial Reference. This is also the Spatial Reference for the GeometryCollection.

GeometryCollection places no other constraints on its elements. Subclasses of GeometryCollection may restrict membership based on dimension and may also place other constraints on the degree of spatial overlap between elements.

2.2.2.1 Methods

Property **NumGeometries** As Long —Returns the number of geometries in the collection.

Function **Geometry**(index As Long) As IGeometry—Returns an indexed geometry in the collection. The indexing starts from zero.

2.2.3 Point

A Point is a 0-dimensional geometry and represents a single location in coordinate space. A point has an x-coordinate value and a y-coordinate value.

The boundary of a point is the empty set.

2.2.3.1 Attributes :

Property **x** As Double —The x-coordinate value for the point.

Property **y** As Double —The y-coordinate value for the point.

Sub **Coords**(x As Double, y As Double) – the X and Y coordinates for the point

2.2.4 MultiPoint

A MultiPoint is a 0 dimensional geometric collection. The elements of a MultiPoint are restricted to Points. The points are not connected or ordered.

A MultiPoint is simple if no two Points in the MultiPoint are equal (have identical coordinate values).

The boundary of a MultiPoint is the empty set.

2.2.5 Curve

A curve is a one-dimensional geometric object usually stored as a sequence of points, with the subtype of curve specifying the form of the interpolation between points. This specification defines only one subclass of curve, LineString, which uses linear interpolation between points.

Topologically a curve is a one-dimensional geometric object that is the homeomorphic image of a real, closed, interval $D = [a, b] = \{x \in R \;/\; a <= x <= b\}$ under a mapping $f:[a,b] \rightarrow \Re^2$ as defined in [1], section 3.12.7.2.

A curve is simple if it does not pass through the same point twice ([1], section 3.12.7.3)

$$\forall c \in Curve, [a, b] = c.Domain,$$

$$c.IsSimple \Leftrightarrow (\forall x1, x2 \in (a, b]\ x1 \neq x2 \Rightarrow f(x1) \neq f(x2)) \wedge (\forall x1, x2 \in [a, b)\ x1 \neq x2 \Rightarrow f(x1) \neq f(x2))$$

A curve is closed if its start point is equal to its end point. ([1], section 3.12.7.3)

The boundary of a closed curve is empty.

A Curve that is simple and closed is a Ring.

The boundary of a non-closed curve consists of its two end points. ([1], section 3.12.3.2).

A Curve is defined as topologically closed.

2.2.5.1 Methods

Property **Length** As Double —The length of the curve in its associated spatial reference.

Function **StartPoint**() As IPoint —The start point of the curve.

Function **EndPoint**() As IPoint —The end point of the curve.

Function **Value**(t As Double) As IPoint—The position of a point on the line, parameterised by length.

Property **IsClosed** As Boolean—Returns TRUE if the curve is closed. The start point and end point of a closed curve are in the same place.

2.2.6 LineString, Line, LinearRing

A LineString is a curve with linear interpolation between points. Each consecutive pair of points defines a line segment.

A Line is a LineString with exactly 2 points.

A LinearRing is a LineString that is both closed and simple. The curve in Figure 2.6—(3) is a closed LineString that is a LinearRing. The curve in Figure 2.6—(4) is a closed LineString that is not a LinearRing.

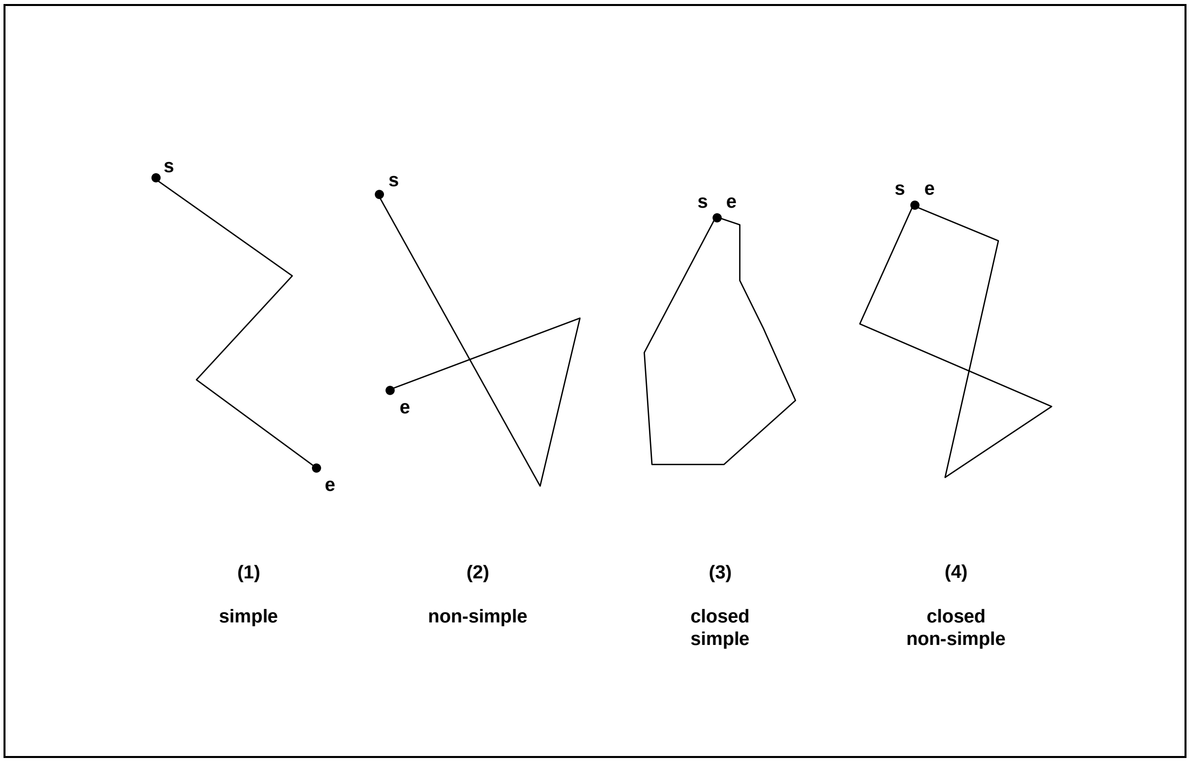


Figure 2.6—(1) a simple LineString, (2) a non-simple LineString, (3) a simple, closed LineString (a LinearRing), (4) a non-simple closed LineString

2.2.6.1 Methods

Property **NumPoints** As Long—The number of points in the LineString.

Function **Point**(index As Long) As IPoint—Returns an indexed point in the LineString. The indexing starts with zero.

2.2.7 MultiCurve

A MultiCurve is a one-dimensional GeometryCollection whose elements are Curves (Figure 2.7).

MultiCurve is a non-instantiable class in this specification, it defines a set of methods for its subclasses and is included for reasons of extensibility.

A MultiCurve is simple if and only if all of its elements are simple and the only intersections between any two elements occur at points that are on the boundaries of both elements.

The boundary of a MultiCurve is obtained by applying the “mod 2” union rule: A point is in the boundary of a MultiCurve if it is in the boundaries of an odd number of elements of the MultiCurve. ([1], section 3.12.3.2).

A MultiCurve is closed if all of its elements are closed. The boundary of a closed multicurve is always empty.

A MultiCurve is defined as topologically closed.

2.2.7.1 Methods

Property **IsClosed** As Boolean—Returns TRUE if the MultiCurve is closed

Property **Length** As Double—The Length of *this* MultiCurve which is equal to the sum of the lengths of the element Curves.

2.2.8 MultiLineString

A MultiLineString is a MultiCurve whose elements are LineStrings.

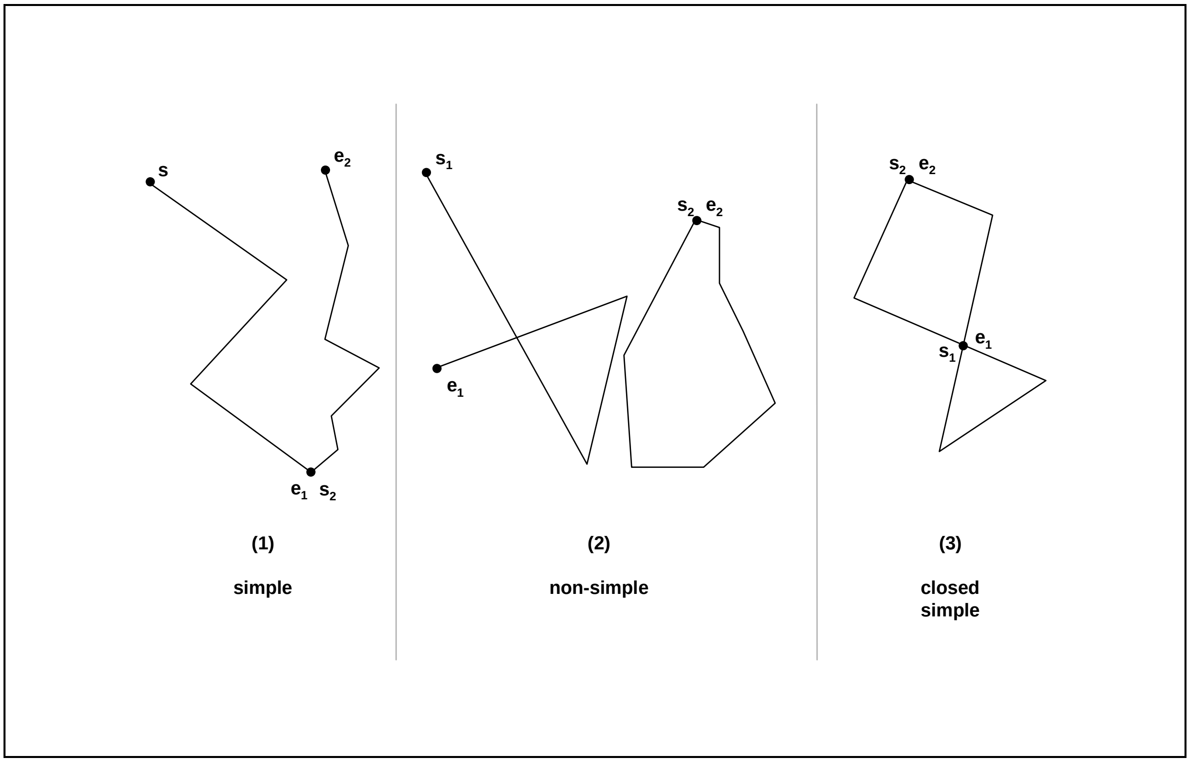


Figure 2.7—(1) a simple MultiLineString, (2) a non-simple MultiLineString with 2 elements, (3) a simple, closed MultiLineString with 2 elements

The boundaries for the MultiLineStrings in Figure 2.7 are (1)—{s1, e2}, (2)—{s1, e1}, (3)—∅

2.2.9 Surface

A Surface is a two-dimensional geometric object.

The OpenGIS Abstract Specification defines a simple surface as consisting of a single ‘patch’ that is associated with one ‘exterior boundary’ and 0 or more ‘interior’ boundaries. Simple surfaces in three-dimensional space are isomorphic to planar surfaces. Polyhedral surfaces are formed by ‘stitching’ together simple surfaces along their boundaries, polyhedral surfaces in three-dimensional space may not be planar as a whole ([1], sections 3.12.9.1, 3.12.9.3).

The boundary of a simple surface is the set of closed curves corresponding to its ‘exterior’ and ‘interior boundaries. ([1], section 3.12.9.4).

The only instantiable subclass of surface defined in this specification, Polygon, is a simple surface that is planar.

2.2.9.1 Methods

Property **Area** As Double—The area of the surface, as measured in its spatial reference system.

Function **Centroid**() As IPoint—The mathematical centroid for the surface. The result is not guaranteed to be on the surface.

Function **PointOnSurface**() As IPoint—A point guaranteed to be on the surface.

2.2.10 Polygon

A Polygon is a planar surface, defined by 1 exterior boundary and 0 or more interior boundaries. Each interior boundary defines a hole in the polygon.

The assertions for polygons (the rules that define valid polygons) are:

1. Polygons are topologically closed.

2. The boundary of a polygon consists of a set of LinearRings that make up its exterior and interior boundaries.

3. No two rings in the boundary cross, the rings in the boundary of a polygon may intersect at a point but only as a tangent :

   $\forall P \in Polygon,\ \forall c1, c2 \in P.Boundary(),\ c1 \neq c2,\ \forall p, q \in Point,\ p, q \in c1,\ p \neq q,\ [\ p \in c2 \Rightarrow q \notin c2]$

4. A Polygon may not have cut lines, spikes or punctures:

   $\forall P \in Polygon,\ P = Closure(Interior(P))$

5. The Interior of every Polygon is a connected point set.

6. The Exterior of a Polygon with 1 or more holes is not connected. Each hole defines a connected component of the Exterior.

In the above assertions, Interior, Closure and Exterior have the standard topological definitions. The combination of 1 and 3 make a Polygon a Regular Closed point set.

Polygons are simple geometries.

Figure 2.8 shows some examples of Polygons. Figure 2.9 shows some examples of geometric objects that violate the above assertions and are not representable as single instances of Polygon. The objects shown in Figure 2.9—(1) and 2.9—(4) can be represented as 2 separate Polygons.

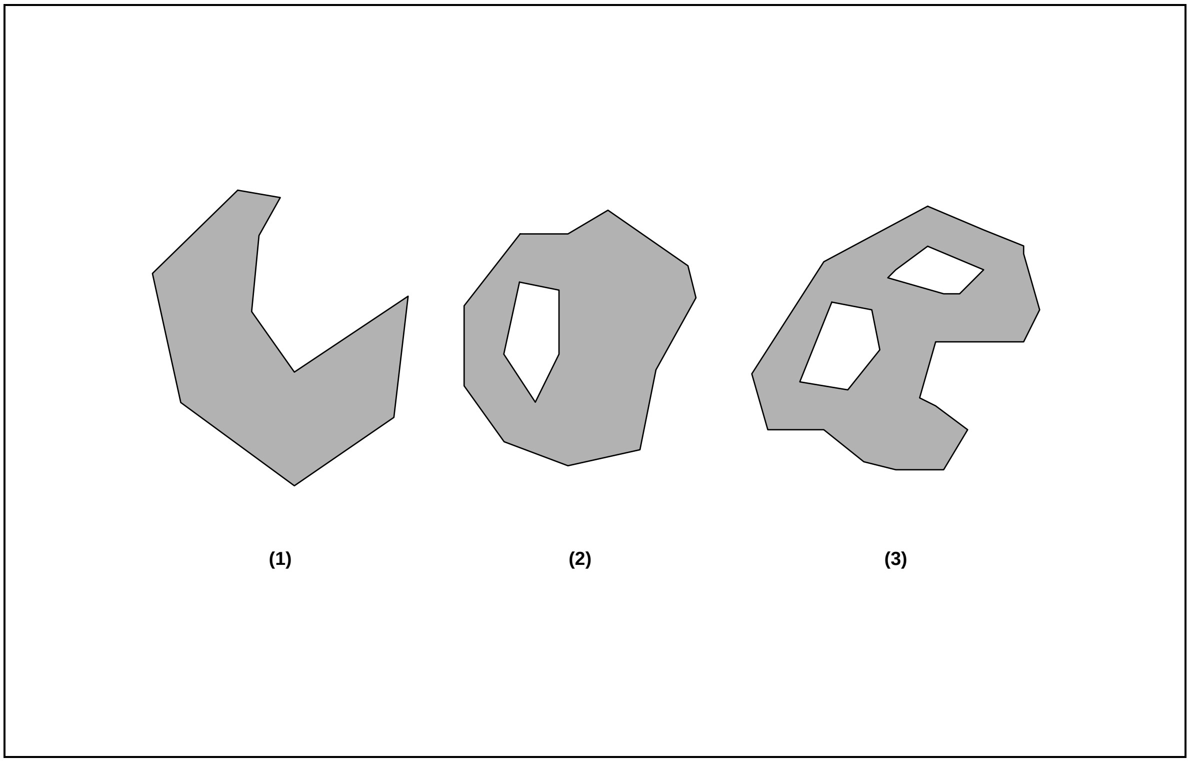


Figure 2.8—Examples of Polygons with 1, 2 and 3 rings respectively .

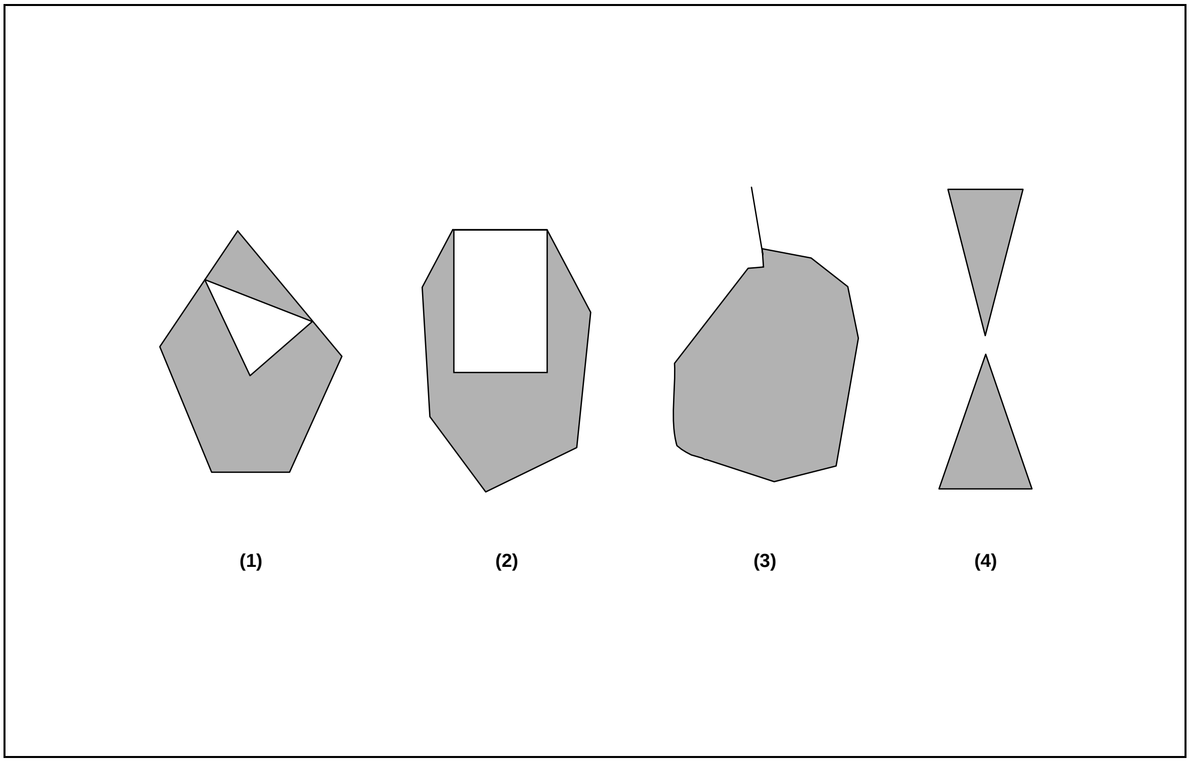


Figure 2.9—Examples of objects not representable as a single instance of Polygon. (1) and (4) can be represented as 2 separate Polygons.

2.2.10.1 Methods

Function **ExteriorRing**() As ILinearRing—Returns the exterior ring of the Polygon.

Property **NumInteriorRings** As Long—Returns the number of interior rings in the Polygon.

Sub **InteriorRing**(index As Long, InteriorRing As ILinearRing) —Returns an indexed interior ring. The indexing starts at zero.

2.2.11 MultiSurface

A MultiSurface is a two-dimensional geometric collection whose elements are surfaces. The interiors of any two surfaces in a MultiSurface may not intersect. The boundaries of any two elements in a MultiSurface may intersect at most at a finite number of points.

MultiSurface is a non-instantiable class in this specification, it defines a set of methods for its subclasses and is included for reasons of extensibility. The instantiable subclass of MultiSurface is MultiPolygon, corresponding to a collection of Polygons.

2.2.11.1 Methods

Property **Area** As Double —The area of the MultiSurface, as measured in its spatial reference system.

Function **Centroid**() As IPoint —The mathematical centroid for the MultiSurface. The result is not guaranteed to be on the MultiSurface.

Function **PointOnSurface**() As IPoint —A point guaranteed to be on the MultiSurface.

2.2.12 MultiPolygon

A MultiPolygon is a MultiSurface whose elements are Polygons..

The assertions for MultiPolygons are :

1. The interiors of 2 Polygons that are elements of a MultiPolygon may not intersect.

   $\forall M \in MultiPolygon, \forall Pi, Pj \in M.Geometries(), i \neq j, Interior(Pi) \cap Interior(Pj) = \emptyset$

2. The Boundaries of any 2 Polygons that are elements of a MultiPolygon may not ‘cross’ and may touch at only a finite number of points. (Note that crossing is prevented by assertion 1 above).

   $\forall M \in MultiPolygon, \forall Pi, Pj \in M.Geometries(), \forall ci \in Pi.Boundaries(), cj \in Pj.Boundaries()$
   $ci \cap cj = \{p1, \ldots.., pk \mid pi \in Point, 1 <= i <= k\}$

3. A MultiPolygon is defined as topologically closed.

4. A MultiPolygon may not have cut lines, spikes or punctures, a MultiPolygon is a Regular, Closed point set:

   $\forall M \in MultiPolygon, M = Closure(Interior(M))$

5. The interior of a MultiPolygon with more than 1 Polygon is not connected, the number of connected components of the interior of a MultiPolygon is equal to the number of Polygons in the MultiPolygon.

The boundary of a MultiPolygon is a set of closed curves (LineStrings) corresponding to the boundaries of its element Polygons. Each curve in the boundary of the MultiPolygon is in the boundary of exactly 1 element Polygon, and every curve in the boundary of an element Polygon is in the boundary of the MultiPolygon.

The reader is referred to work by Worboys, et. al (7, 8) and Clementini, et. al (5, 6) for work on the definition and specification of MultiPolygons.

Figure 2.10 shows 4 examples of valid MultiPolygons with 1, 3, 2 and 2 polygon elements respectively.

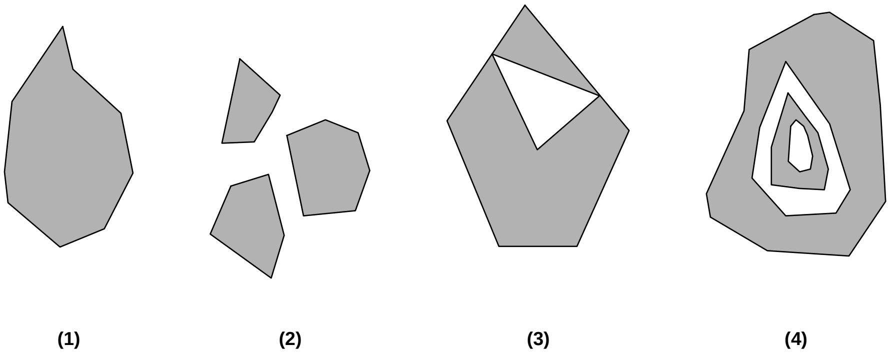


Figure 2.10—Examples of MultiPolygons

Figure 2.11 shows examples of geometric objects not representable as single instances of MultiPolygons.

Note that the subclass of Surface named Polyhedral Surface described in the [1], is a faceted surface whose facets are Polygons. A Polyhedral Surface is not a MultiPolygon because it violates the rule for MultiPolygons that the boundaries of the element Polygons intersect only at a finite number of points.

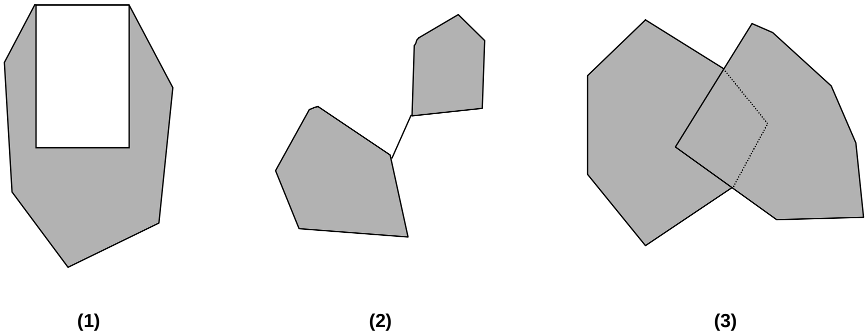


Figure 2.11—Geometric objects not representable as a single instance of a MultiPolygon.

2.2.13 Relational Operators

This section provides a more detailed specification of the relational operators on geometries.

2.2.13.1 Background

The Relational Operators are Boolean methods that are used to test for the existence of a specified topological spatial relationship between two geometries. Topological spatial relationships between two geometric objects have been a topic of extensive study in the literature [4,5,6,7,8,9,10]. The basic approach to comparing two geometries is to make pair-wise tests of the intersections between the Interiors, Boundaries and Exteriors of the two geometries and to classify the relationship between the two geometries based on the entries in the resulting 'intersection' matrix.

The concepts of Interior, Boundary and Exterior are well defined in general topology. For a review of these concepts the user is referred to Egenhofer, et al [4]. These concepts can be applied in defining spatial relationships between two-dimensional objects in two-dimensional space ($\Re^2$). In order to apply the concepts of Interior, Boundary and Exterior to 1 and 0 dimensional objects in $\Re^2$, a combinatorial topology approach must be applied. ([1], section. 3.12.3.2). This approach is based on the accepted definitions of the boundaries, interiors and exteriors for simplicial complexes [12] and yields the following results:

The boundary of a geometry is a set of geometries of the next lower dimension. The boundary of a Point or a MultiPoint is the empty set. The boundary of a non-closed Curve consists of its two end Points, the boundary of a closed Curve is empty. The boundary of a MultiCurve consists of those Points that are in the boundaries of an odd number of its element Curves. The boundary of a Polygon consists of its set of Rings. The boundary of a MultiPolygon consists of the set of Rings of its Polygons. The boundary of an arbitrary collection of geometries whose interiors are disjoint consists of geometries drawn from the boundaries of the element geometries by application of the "mod 2" union rule ([1], section 3.12.3.2).

The domain of geometric objects considered is those that are topologically closed. The interior of a geometry consists of those points that are left when the boundary points are removed. The exterior of a geometry consists of points not in the interior or boundary.

Studies on the relationships between two geometries both of maximal dimension in $\Re^1$ and $\Re^2$ considered pair-wise intersections between the Interior and Boundary sets and led to the definition of a 4 Intersection Model [8]. The model was extended to consider the exterior of the input geometries, resulting in a nine intersection model [11] and further extended to include information on the dimension of the results of the pair-wise intersections resulting in a dimensionally extended nine intersection model [5]. These extensions allow the model to express spatial relationships between points, lines and areas, including areas with holes and multi component lines and areas [6].

2.2.13.2 The Dimensionally Extended Nine-Intersection Model

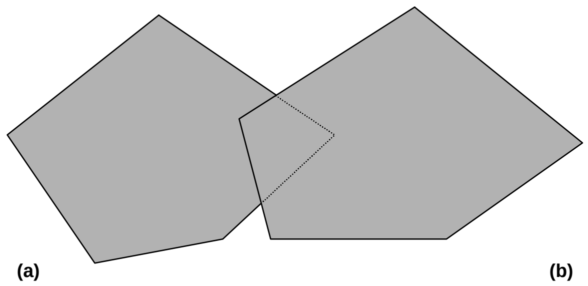
Given a geometry a, let *I(a)*, *B(a)* and *E(a)* represent the Interior, Boundary and Exterior of a respectively. The intersection of any two of *I(a)*, *B(a)* and *E(a)* can result in a set of geometries, *x*, of mixed dimension. For example, the intersection of the boundaries of two polygons may consist of a point and a line. Let *dim(x)* return the maximum dimension (-1, 0, 1, or 2) of the geometries in *x*, with a numeric value of -1 corresponding to $dim(\emptyset)$. A dimensionally extended nine-intersection matrix (DE-9IM) then has the form:

	Interior	Boundary	Exterior
Interior	$dim(I(a)\cap I(b))$	$dim(I(a)\cap B(b))$	$dim(I(a)\cap E(b))$
Boundary	$dim(B(a)\cap I(b))$	$dim(B(a)\cap B(b))$	$dim(B(a)\cap E(b))$
Exterior	$dim(E(a)\cap I(b))$	$dim(E(a)\cap B(b))$	$dim(E(a)\cap E(b))$

Table 2.2—The DE-9IM

For regular, topologically closed input geometries, computing the dimension of the intersection of the Interior, Boundary and Exterior sets does not have as a prerequisite the explicit computation and representation of these sets. For example to compute if the interiors of two regular closed polygons intersect, and to ascertain the dimension of this intersection, it is not necessary to explicitly represent the interior of the two polygons (which are topologically open sets) as separate geometries. In most cases the dimension of the intersection value at a cell is highly constrained given the type of the two geometries. For example, in the Line-Area case the only possible values for the Interior-Interior cell are drawn from {-1, 1} and in the Area-Area case the only possible values for the Interior-Interior cell are drawn from {-1, 2}. In such cases no work beyond detecting the intersection is required.

Figure 2.12 shows an example DE-9IM for the case where *a* and *b* are two polygons that overlap.



	Interior	Boundary	Exterior
Interior	2	1	2
Boundary	1	0	1
Exterior	2	1	2

Figure 2.12—An example instance and its DE-9IM

A spatial relationship predicate can be formulated on two geometries that takes as input a pattern matrix representing the set of acceptable values for the DE-9IM for the two geometries. If the spatial relationship between the two geometries corresponds to one of the acceptable values as represented by the pattern matrix, then the predicate returns `TRUE`.

The pattern matrix consists of a set of 9 pattern-values, one for each cell in the matrix. The possible pattern-values *p* are {T, F, *, 0, 1, 2} and their meanings for any cell where *x* is the intersection set for the cell are as follows:

$p = T \Rightarrow dim(x) \in \{0, 1, 2\}$, i.e. $x \neq \emptyset$

$p = F \Rightarrow dim(x) = -1$, i.e. $x = \emptyset$

$p = * \Rightarrow dim(x) \in \{-1, 0, 1, 2\}$, i.e. Don't Care

$p = 0 \Rightarrow dim(x) = 0$

$p = 1 \Rightarrow dim(x) = 1$

$p = 2 \Rightarrow dim(x) = 2$

The pattern matrix can be represented as an array or list of nine characters in row major order. As an example the following code fragment could be used to test for "Overlap" between two areas:

```
char * overlapMatrix = “T*T***T**”;
Geometry* a, b;
Boolean b = a->Relate(b, overlapMatrix);
```

2.2.13.3 Named Spatial Relationship predicates based on the DE-9IM

The Relate predicate based on the pattern matrix has the advantage that clients can test for a large number of spatial relationships and fine tune the particular relationship being tested. It has the disadvantage that it is a lower level building block and does not have a corresponding natural language equivalent. Users of the proposed system include IT developers using the COM API from a language such as Visual Basic, and interactive SQL users who may wish, for example, *to select all features 'spatially within' a query polygon*, in addition to more spatially 'sophisticated' GIS developers.

To address the needs of such users a set of named spatial relationship predicates have been defined in [5,6] for the DE-9IM. The five predicates are named Disjoint, Touch, Cross, In and Overlap and have the following properties:

1. They are mutually exclusive.

2. They provide a complete covering of all topological cases.

3. They apply to spatial relationships between two geometries of either the same or different dimension.

4. Each predicate can be expressed in terms of a corresponding set of DE-9IM matrix patterns.

5. Any realizable DE-9IM can be expressed as a boolean expression over the 5 predicates, given the Boundary method on Geometry and the StartPoint and EndPoint method on Curve.

The definition of these predicates [5,6] is given below. In these definitions the term P is used to refer to 0 dimensional geometries (Points and MultiPoints), L is used to refer to one-dimensional geometries (LineStrings and MultiLineStrings) and A is used to refer to two-dimensional geometries (Polygons and MultiPolygons).

Disjoint

Given two (topologically closed) geometries *a* and *b*,

$$a.Disjoint(b) \Leftrightarrow a \cap b = \emptyset$$

Expressed in terms of the DE-9IM:

$$a.Disjoint(b) \Leftrightarrow (I(a) \cap I(b) = \emptyset) \wedge (I(a) \cap B(b) = \emptyset) \wedge (B(a) \cap I(b) = \emptyset) \wedge (B(a) \cap B(b) = \emptyset)$$
$$\Leftrightarrow a.Relate(b, \text{“FF*FF****”})$$

Touches

The Touches relation between two geometries a and b applies to the A/A, L/L, L/A, P/A and P/L groups of relationships but not to the P/P group. It is defined as:

$$a.Touches(b) \Leftrightarrow (I(a) \cap I(b) = \emptyset) \wedge (a \cap b) \neq \emptyset$$

Expressed in terms of the DE-9IM:

$$a.Touches(b) \Leftrightarrow (I(a) \cap I(b) = \emptyset) \wedge ((B(a) \cap I(b) \neq \emptyset) \vee (I(a) \cap B(b) \neq \emptyset) \vee (B(a) \cap B(b) \neq \emptyset))$$
$$\Leftrightarrow a.Relate(b, \text{“FT*******”}) \vee a.Relate(b, \text{“F**T*****”}) \vee a.Relate(b, \text{“F***T****”})$$

Figure 2.13 shows some examples of the Touches relation.

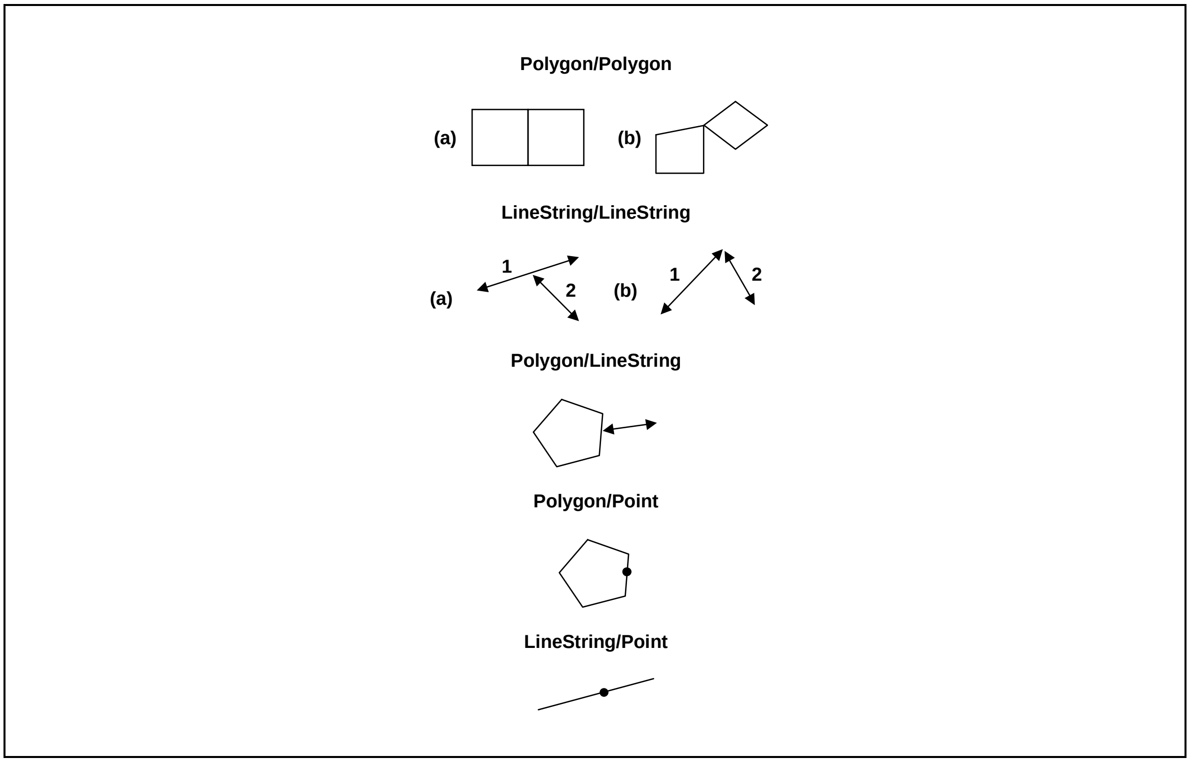


Figure 2.13—Examples of the Touch relationship

Crosses

The Crosses relation applies to P/L, P/A, L/L and L/A situations. It is defined as:

$$a.Crosses(b) \Leftrightarrow (dim(I(a) \cap I(b) < max(dim(I(a)), dim(I(b)))) \wedge (a \cap b \neq a) \wedge (a \cap b \neq b)$$

Expressed in terms of the DE-9IM:

Case $a \in P, b \in L$ or Case $a \in P, b \in A$ or Case $a \in L, b \in A$:

$$a.Crosses(b) \Leftrightarrow (I(a) \cap I(b) \neq \emptyset) \wedge (I(a) \cap E(b) \neq \emptyset) \Leftrightarrow a.Relate(b, \text{“T*T******”})$$

Case $a \in L, b \in L$:

$$a.Crosses(b) \Leftrightarrow dim(I(a) \cap I(b)) = 0 \Leftrightarrow a.Relate(b, \text{“0********”});$$

Figure 2.14 shows some examples of the Crosses relation.

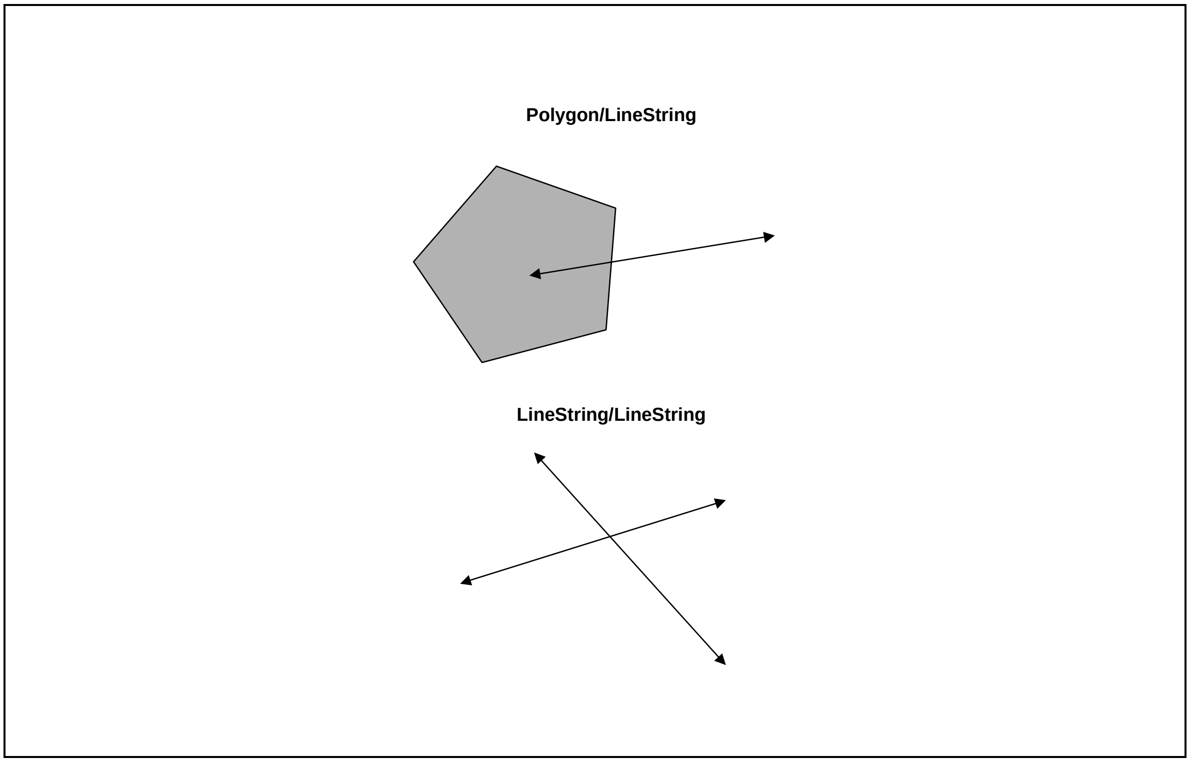


Figure 2.14—Examples of the Cross relationship

In (Within)

The Within relation is defined as:

$$a.Within(b) \Leftrightarrow (a \cap b = a) \wedge (I(a) \cap E(b) \neq \emptyset)$$

Expressed in terms of the DE-9IM:

$$a.Within(b) \Leftrightarrow (I(a) \cap I(b) \neq \emptyset) \wedge (I(a) \cap E(b) = \emptyset) \wedge (B(a) \cap E(b) = \emptyset)) \Leftrightarrow a.Relate(b, \text{“TF*F*****”})$$

Figure 2.15 shows some examples of the Within relation.

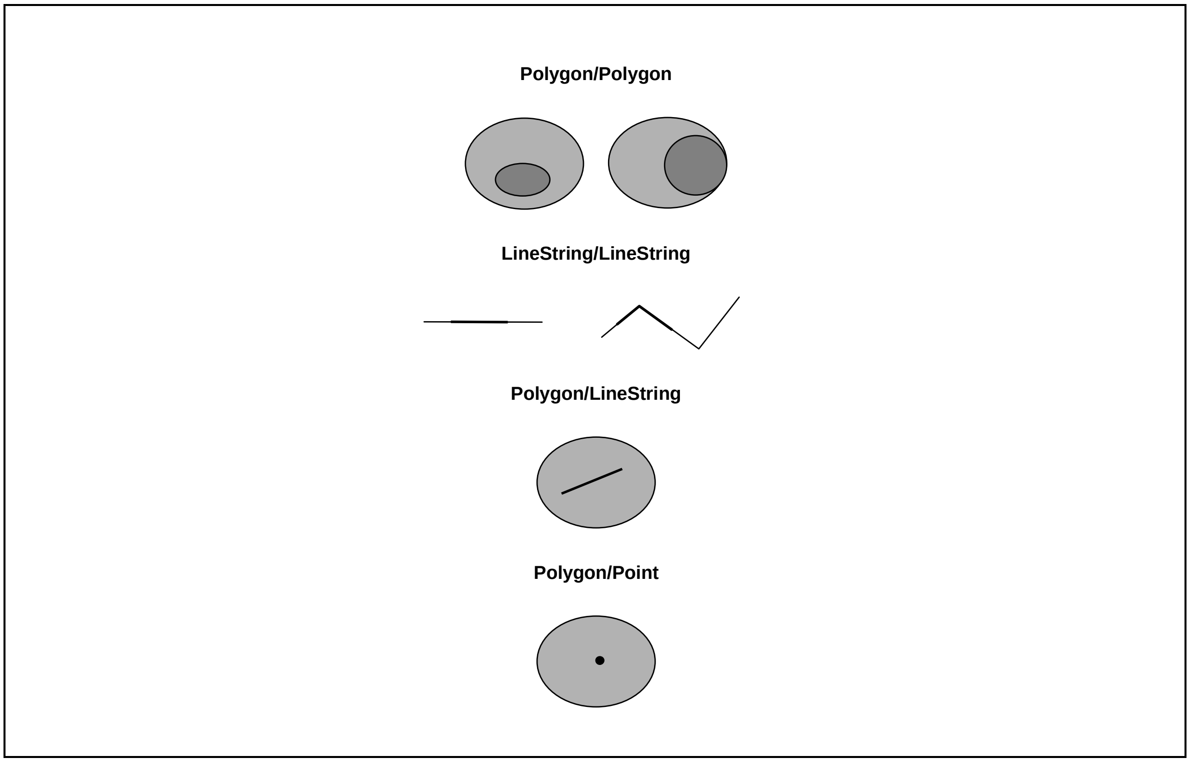


Figure 2.15—Examples of the Within relationship

Overlaps

The Overlaps relation is defined for A/A, L/L and P/P situations.

It is defined as:

$$a.Overlaps(b) \Leftrightarrow (dim(I(a)) = dim(I(b)) = dim(I(a) \cap I(b))) \wedge (a \cap b \neq a) \wedge (a \cap b \neq b)$$

Expressed in terms of the DE-9IM:

Case $a \in P, b \in P$ or Case $a \in A, b \in A$:

$$a.Overlaps(b) \Leftrightarrow (I(a) \cap I(b) \neq \emptyset) \wedge (I(a) \cap E(b) \neq \emptyset) \wedge (E(a) \cap I(b) \neq \emptyset) \Leftrightarrow a.Relate(b, \text{“T*T***T**”})$$

Case $a \in L, b \in L$:

$$a.Overlaps(b) \Leftrightarrow (dim(I(a) \cap I(b) = 1) \wedge (I(a) \cap E(b) \neq \emptyset) \wedge (E(a) \cap I(b) \neq \emptyset) \Leftrightarrow a.Relate(b, \text{“1*T***T**”})$$

Figure 2.16 shows some examples of the Overlaps relation.

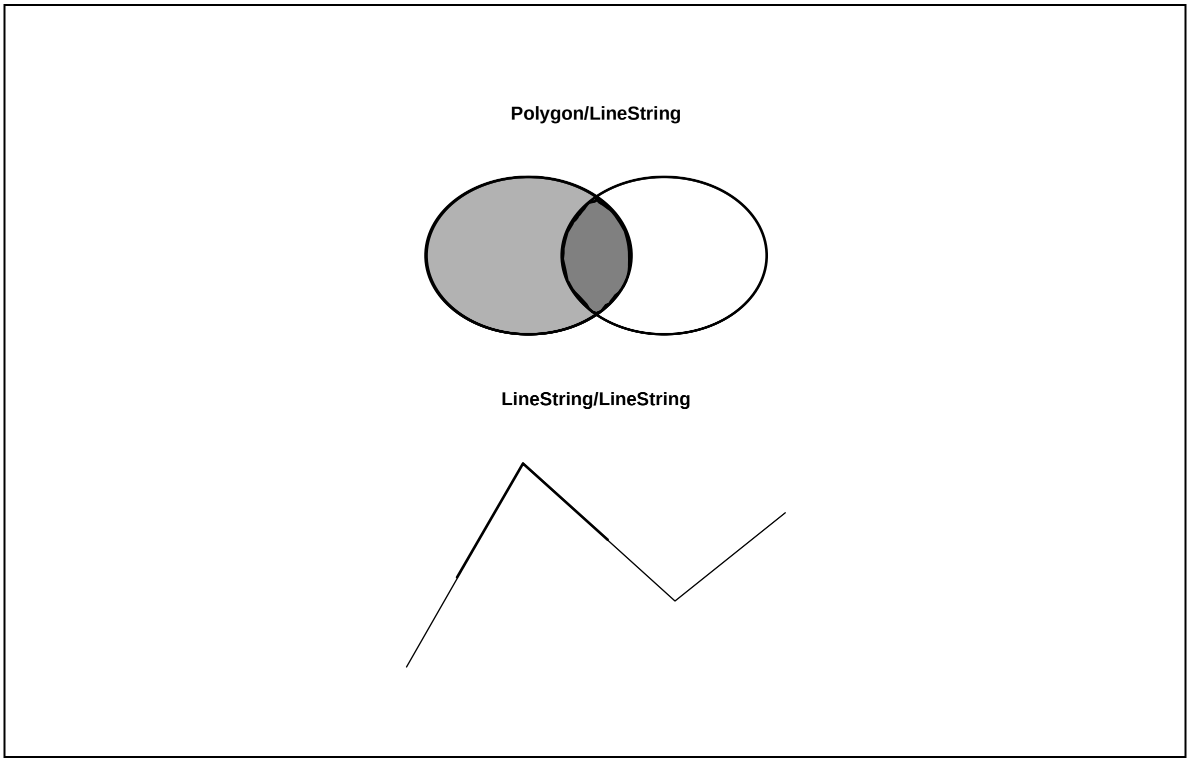


Figure 2.16—Examples of the Overlap relationship

The following additional named predicates are also defined for user convenience:

Contains

$$a.Contains(b) \Leftrightarrow b.Within(a)$$

Intersects

$$a.Intersects(b) \Leftrightarrow !\, a.Disjoint(b)$$

2.3 Spatial Reference System Object Model

The Spatial Reference System Object Model proposed in this specification is shown in Figure 2.17. This object model is based upon the OpenGIS Abstract Specification and uses the geodetic model for spatial reference systems of the European Petroleum Survey Group (EPSG) and the Petrotechnical Open Software Corp. (POSC). Chapter 3 of this document describes the set of COM Classes and Interfaces that implement this object Model.

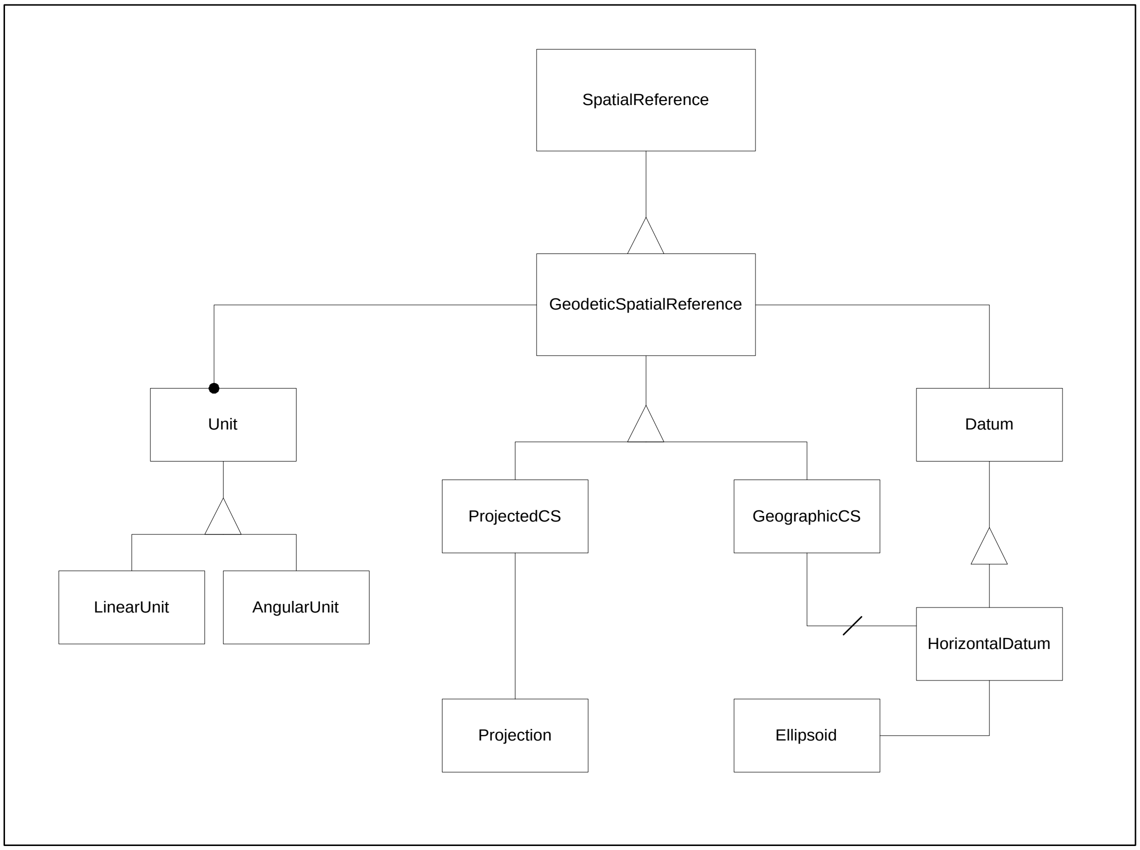


Figure 2.17—The Spatial Reference System Object Model

2.4 Summary

The architecture and details contained in this specification satisfy the functional requirements of this specification subject to the detail notes in the narrative. The architecture has the following characteristics:

- Provides a COM OLE implementation consistent with OLE DB
- Easily implemented by multiple vendors
- Designed with performance in mind
- Designed with reliability and data integrity in mind
- Testable implementations
- Extensible in all key areas

2.4.1 Requirement summary

This specification describes data access methodologies, geometry objects, and spatial reference objects. Vendors, consultants and users are not required to implement the complete suite of objects. It is expected that implementations from a variety of users will be mixed in a given application environment. For example, a consultant may utilize data providers from Vendors A, B and C. He can then use Vendor D's geometry objects to analyze the geometry extracted from the data sources.

3 Component Specifications

3.1 OLEDB and ADO Components

ADO and OLEDB are the facilities by which consumers and providers communicate data and metadata. To achieve a sense of seamless interoperability, standards must be defined to allow GIS consumers and providers to communicate GIS information using these facilities. These standards enable an OLEDB data consumer to determine the GIS capabilities of an OLEDB data provider and retrieve GIS information in a predictable manner.

An OLEDB data consumer that cares nothing about GIS may still utilize data from an OGIS data provider as if it were any other OLEDB data provider. This consumer will simply not take advantage of the GIS information. Similarly, an OGIS data consumer can use the same OLEDB API, without the GIS standards, to utilize data from any OLEDB data provider.

These standards involve:

- OGIS Data Provider Registry Entries—OGIS data providers must register support for the `"OGISDataProvider"` component category so that consumers can distinguish them from other OLEDB Data Providers.

- GIS Metadata—Required and Optional GIS Metadata and the manner by which the client retrieves it are described. To a great extent, this is the bulk of the proposal—this is how the client knows there is GIS information and how to get it.

    - Which tables are considered GIS features

    - Which columns contain geometry; what the geometry's spatial reference is and what type of geometry column it is.

    - What the Spatial Reference(s) of the Data Source are.

    - What Spatial Operators are supported by the OLEDB Data Provider

- `IColumnsRowset`—Additional GIS columns are defined for the `Rowset` returned by the `GetColumnsRowset` method.

- Geometry—Methods are described for acquiring geometry from a Column or Field as a WKBGeometry in OLEDB or a Variant in ADO.

- Spatial Reference Information—Methods are described for acquiring Spatial Reference information from the Session level as well as from the Rowset as a Variant in OLEDB or in ADO.

- Spatial Filter—Standard spatial filter parameters are defined for use with a Command in ADO and in OLEDB. Parameters are spatial filter, spatial operator and Geometry Field/Column name.

Although these standards are needed to allow GIS consumers and providers to communicate GIS information, there is no strict "level of compliance" scheme for OGIS data servers. Clients are expected to use standard COM and OLE DB techniques for determining if a data server supports a specified type of functionality. It is up to the client to decide what action it will take if a provider does not provide that support.

The minimum level of support that a data server must pass is registering support for the "OGISDataProvider" component category.

Data providers which do not support schema rowsets or IColumnsRowset::GetColumnsRowset are encouraged to name their geometry columns "OGIS_GEOMETRY". Clients are also encouraged to look for columns with this name if the provider does not support the schema methods noted above.

A data provider which wishes to provide additional GIS Metadata and Geometry information to the client, should do so in compliance with the following sections.

3.1.1 OGIS Data Provider Registry Entries

OGIS OLEDB Data Providers must register support for the "OGISDataProvider" component category. Its GUID is CATID_OGISDataProvider[1]. Consumers can use this to distinguish OGIS OLEDB Data Providers.

3.1.2 GIS Metadata

When an OLEDB data provider exposes GIS Metadata, the consumer can subsequently access and interpret data in a GIS context. This metadata is three additional SchemaRowsets and an additional PropertySet.

3.1.3 DBSCHEMA_OGIS_FEATURE_TABLES Rowset

This rowset indicates those tables that the consumer can query as features. Any entry in the DBSCHEMA_OGIS_FEATURE_TABLES Rowset also appears as an entry in the standard OLEDB Tables Rowset[2].The columns in the two rowsets are different, but the rows in the standard Tables Rowset should be a superset of the rows in the DBSCHEMA_OGIS_FEATURE_TABLES Rowset.. Although there is a column for FEATURE_TABLE_ALIAS which might be exposed in a Graphical User Interface, this is not what should be used while querying. TABLE_NAME along with TABLE_CATALOG and TABLE_SCHEMA should be used when performing database queries.

Column_Name	**Type_Indicator**	**Description**

[1] The CATID_OGISDataProvider GUID is defined in OLEDBGIS.h, supplied separately

[2] This specification makes no restriction on Table Type and assumes appropriate use of OLE/DB schema rowsets.

FEATURE_TABLE_ALIAS	DBTYPE_WSTR	User Friendly Feature Name—may be NULL
TABLE_CATALOG	DBTYPE_WSTR	Catalog name in which the table is defined. NULL if the provider does not support catalogs.
TABLE_SCHEMA	DBTYPE_WSTR	Schema name in which the Feature Table is defined, NULL if the provider does not support schemas.
TABLE_NAME	DBTYPE_WSTR	Feature Table Name
ID_COLUMN_NAME	DBTYPE_WSTR	Preferred column name to reference rows. OGIS requires this column to have a name[3].
DG_COLUMN_NAME	DBTYPE_WSTR	Default Geometry column name. OGIS requires this column to have a name.

This rowset can be accessed in OLEDB from the Session via IDBSchemaRowset::GetRowset, passing the GUID (OGIS_FEATURE_TABLES_GUID) for the DBSCHEMA_OGIS_FEATURE_TABLES Rowset. In ADO this is achieved via Connection.OpenSchema (enum, [restrictions], [guid]). The enum value in this case will be a specific ADO enum[4] indicating that the last argument is a GUID that must be supplied by the caller and that GUID is what is passed to the OLE-DB provider. In this case it is the OGIS_FEATURE_TABLES_GUID[5].

There are no restrictions defined on the DBSCHEMA_OGIS_FEATURE_TABLES Rowset.

3.1.4 DBSCHEMA_OGIS_GEOMETRY_COLUMNS Rowset

This rowset identifies the feature columns in the catalog which are geographic geometry. The feature identified by TABLE_CATALOG, TABLE_SCHEMA, and TABLE_NAME must appear in the DBSCHEMA_OGIS_FEATURE_TABLES Rowset. The geometry type and spatial reference system are specified for the column.

Column_Name	Type_Indicator	Description
TABLE_CATALOG	DBTYPE_WSTR	Catalog name in which the Feature’s Table is defined. NULL if the provider does not support catalogs.
TABLE_SCHEMA	DBTYPE_WSTR	Schema name in which the Feature’s Table is defined, NULL if the provider does not support schemas.
TABLE_NAME	DBTYPE_WSTR	The Feature Table Name.
COLUMN_NAME	DBTYPE_WSTR	Name of column containing geometry

[3] The use of just COLUMN_GUID and COLUMN_PROPID is not sufficient for OGIS.

[4] The ADO-supplied enum for this will be available with ADO 1.5.

[5] The OGIS_FEATURE_TABLES_GUID is defined in OLEDBGIS.h, supplied separately.

GEOM_TYPE	DBTYPE_UI4	Type of geometry column. Values taken from the OGIS_Geometry Enumerated Type.
SPATIAL_REF_SYSTEM_ID	DBTYPE_I4	Foreign Key—this is the ID of the Spatial Reference System of the geometry column. This ID can be used to find the Spatial Reference in the DBSCHEMA_OGIS_SPATIALREFERENCESYSTEMS Rowset

This rowset can be accessed from the Session via IDBSchemaRowset::GetRowset, passing the GUID (DBSCHEMA_OGIS_GEOMETRY_COLUMNS) for the DBSCHEMA_OGIS_GEOMETRY_COLUMNS Rowset. In ADO this is achieved via Connection.OpenSchema (enum, [restrictions], [guid]). The enum value in this case will be a specific ADO enum indicating that the last argument is a GUID that must be supplied by the caller and that GUID is what is passed to the OLE-DB provider. In this case it is the DBSCHEMA_OGIS_GEOMETRY_COLUMNS_GUID[6].

There are no restrictions defined on the DBSCHEMA_OGIS_GEOMETRY_COLUMNS Rowset.

3.1.5 DBSCHEMA_OGIS_SPATIAL_REF_SYSTEMS Rowset

This Rowset indicates the Spatial Reference Systems supported by the data provider in this Session. There can be more than one row in the Rowset. This Rowset contains the set of all the Spatial Reference Systems encountered for all the columns found in the DBSCHEMA_OGIS_GEOMETRY_COLUMNS Rowset.

Column_Name	**Type_Indicator**	**Description**
SPATIAL_REF_SYSTEM_ID	DBTYPE_I4	ID of the Spatial Reference System. May be NULL only if SPATIAL_REF_SYSTEM_WKT is NULL.
AUTHORITY_NAME	DBTYPE_WSTR	Defining Authority for this Spatial Reference System, e.g., "POSC", "USGS". May be NULL.
AUTHORITY_ID	DBTYPE_I4	Authority specific identifier. This is a Well-known id assigned to the spatial reference system by the authority. May be NULL.
SPATIAL_REF_SYSTEM_WKT	DBTYPE_BSTR	The Well-known Text Representation of the Spatial Reference System. May be NULL.

This rowset can be accessed from the Session via IDBSchemaRowset::GetRowset, passing the GUID (DBSCHEMA_OGIS_SPATIAL_REFERENCE) for the DBSCHEMA_OGIS_SPATIAL_REF_SYSTEMS Rowset. In ADO this is achieved via Connection.OpenSchema (enum, [restrictions], [guid]). The enum value in this case will be a specific ADO enum indicating that the last argument is a GUID that must be supplied by the caller and that GUID is what is passed to the OLE-DB provider. In this case it is the DBSCHEMA_OGIS_SPATIAL_REFERENCE[7].

[6] The DBSCHEMA_OGIS_GEOMETRY_COLUMNS is defined in OLEDBGIS.h, supplied separately.

[7] The DBSCHEMA_OGIS_SPATIAL_REF_SYSTEMS is defined in OLEDBGIS.h, supplied separately.

There are no restrictions defined on the DBSCHEMA_OGIS_SPATIAL_REF_SYSTEMS Rowset.

3.1.6 OGIS Property Set

This is a property set for OGIS specific attributes of a Data Source. This property set has GUID DPPROPSET_OGIS_SPATIAL_OPS

NOTE: Need to add two columns to table. First new column is the short description, second column is READ or READ/WRITE flag.

Property ID[8]	**Type Indicator**	**Description**
DBPROP_OGIS_TOUCHES	VT_BOOL	All points in the intersection of geometries of Data Source and the Spatial Filter lie on a geometry boundary and the interiors of the geometries of the Data Source and the Spatial Filter do not intersect
DBPROP_OGIS_WITHIN	VT_BOOL	Geometries of the Data Source are wholly contained by the Spatial Filter.
DBPROP_OGIS_CONTAINS	VT_BOOL	The Spatial Filter is wholly contained by geometries of the Data Source.
DBPROP_OGIS_CROSSES	VT_BOOL	Geometries of the Data Source and the Spatial Filter intersect, but do not wholly contain each other, and the dimension of the intersection of their interiors is one less than the maximum dimension of their interiors.
DBPROP_OGIS_OVERLAPS	VT_BOOL	Geometries of the Data Source and the Spatial Filter intersect and the dimension of the intersection is the same as that of the input geometries but the intersection is different than the input geometries.
DBPROP_OGIS_DISJOINT	VT_BOOL	Intersection of geometries of the Data Source and the Spatial Filter is the empty set.
DBPROP_OGIS_INTERSECTS	VT_BOOL	Intersection of geometries of the Data Source and the Spatial Filter is not the empty set.
DBPROP_OGIS_ENVELOPE_INTERSECTS	VT_BOOL	Intersection of the envelope of geometries of the Data Source and the envelope of the Spatial Filter is not the empty set.
DBPROP_OGIS_INDEX_INTERSECTS	VT_BOOL	Intersection of the spatial index entries of the

[8] Values of Property IDs of the OGIS Property Set are defined in OLEDBGIS.h, supplied separately.

		geometries of the Data Source and the geometry of the Spatial Filter is not the empty set.

3.1.7 IColumnsRowset:GetColumnsRowset

The consumer can access schema rowset information from the Session level via IDBSchemaRowset::GetRowset. However, given a Rowset, the consumer can get specific information about that Rowset via IColumnsRowset::GetColumnsRowset without reverting to the Session.

The standard columns in the IColumnsRowset are as defined by the OLEDB specification.

The OGIS Rowset consumer requires more columns than the standard IColumnsRowset or the IColumnsInfo interfaces can provide. The IColumnsRowset::GetColumnsRowset must additionally provide all columns from the DBSCHEMA_OGIS_GEOMETRY_COLUMNS Rowset defined earlier except for TABLE_CATALOG, TABLE_SCHEMA, TABLE_NAME, and COLUMN_NAME. Currently, that includes GEOM_TYPE and SPATIAL_REF_SYSTEM_ID. In addition, the OGIS OLEDB data provider is obligated to provide the SPATIAL_REF_SYSTEM_WKT from the DBSCHEMA_OGIS_SPATIALREFERENCESYSTEM Rowset that corresponds to the SPATIAL_REF_SYSTEM_ID if the consumer requests it. The consumer requests this optional column by specifying it in rgOptColumns.

For the rows corresponding to non-geometry columns the values of GEOM_TYPE and SPATIAL_REF_SYSTEM_ID (and optional SPATIAL_REF_SYSTEM_WKT) will be NULL.

This information is available to the ADO user by accessing Property objects in the Properties collection for each Field off of the Recordset. If a Field has a non-NULL Property with name GEOM_TYPE, then it contains geometry.

3.1.8 Geometry

The consumer can determine which columns contain geometry by calling IDBSchemaRowset::GetRowset or IColumnsRowset::GetColumnsRowset. Geometry columns should be accessible by binding to the column as DBTYPE_BYTES or as DBTYTPE_IUNKNOWN. In the latter case the requested IID must be either IID_IStream or IID_ISequentialStream.

The consumer can access geometry in a variety of ways

- C++:
  - build an Accessor with a DBBINDING Structure specifying a wType as DBTYPE_BYTES or DBTYPE_BYTES | DBTYPE_BYREF and use IRowset::GetData to access the geometry as a Well-known Binary Representation of Geometry or WKBGeometry.
- ADO:
  - dim a variable as Variant and retrieve the Well-known Binary Representation of Geometry (WKBGeometry) via

```
variable = Field.GetChunk
```

In this situation ADO binds to the geometry column as DBTYPE_IUNKNOWN and requests the IStream interface. It reads the data with this interface to produce a Variant of type VT_ARRAY|VT_UI1.

- dim a variable as Variant and retrieve the WKBGeometry via

  ```
  variable = Field.Value
  ```

In this case ADO binds to the geometry column as DBTYPE_BYTES and reads the data to build a Variant of type VT_ARRAY|VT_UI1[9].

3.1.9 Spatial Reference

The consumer can determine what Spatial References are in the data source by calling IDBSchemaRowset::GetRowset. The consumer can determine the Spatial Reference System of a geometry column of a Rowset by calling IColumnsRowset::GetColumnsRowset. The Spatial_Reference_System_WKT should be accessible by binding to the column as DBTYPE_BSTR.

The consumer can access the values in the Spatial_Reference_System_WKT column in a variety of ways

- C++:
  - build an Accessor with a DBBINDING Structure specifying a wType as DBTYPE_BSTR and use IRowset::GetData to access the geometry as a Well-known Text Representation of SpatialReference (WKTSpatialReference).
- ADO:
  - dim a variable as String or Variant and retrieve the WKTSpatialReference via

    ```
    variable = Field.Value
    ```

In this case ADO binds to the geometry column as DBTYPE_BSTR.

3.1.10 Spatial Filter

OLEDB data providers may allow queries to be spatially filtered. Consumers specify spatial filtering criteria by setting parameters on the Command before execution. There are three parameters for spatial filtering:

Parameter Name	Type	Description
SPATIAL_FILTER	DBTYPE_VARIANT	Variant of type VT_ARRAY|VT_UI1 or VT_UNKNOWN. This is the Well-known Binary Representation of Geometry containing the geometry of the Spatial Filter. In the latter case, the supplied interface pointer must be convertible to

[9] Be acquainted with Chapter 6, “Getting and Setting Data”, in the OLEDB Programmer’s Reference when binding with wType as DBTYPE_BYTES|DBTYPE_BYREF; especially “Data Parts—Length—Length Values”, “DBBINDING Structures—dbMemOwner” and “Memory Management”. When using DBTYPE_BYREF and not copying data into the consumer’s buffer, dbMemOwner governs how memory for the data is freed and its lifetime from the provider and consumer point of view. Also be acquainted with Chapter 7, “BLOBs and OLE Objects” section “BLOBs as In-Memory Data” as well as “Appendix A Data Types” where it describes DBTYPE_BYTES.

		IStream or ISequentialStream.
SPATIAL_OPERATOR	DBTYPE_UI4	Property ID of the spatial operator. OGIS Property Set contains the operators supported by this Data Source.
SPATIAL_GEOM_COL_NAME	DBTYPE_WSTR	Name of column to be spatially filtered.

These parameters are independent of the command language text (e.g. SQL). If the command language text is parameterized, then the spatial parameters appear after any command language parameters. The order of the Spatial Filter parameters is [SPATIAL_FILTER, SPATIAL_OPERATOR, SPATIAL_GEOM_COL_NAME].

3.1.11 OGIS_Geometry Enumerated Type

The enumeration value identifies the geometry type and appears as the GEOM_TYPE of the DBSCHEMA_OGIS_GeometryColumns Rowset. A geometry column may contain geometry of any subclass of the type indicated by the value of GEOM_TYPE.

Value[10]
DBGEOM_GEOMETRY
DBGEOM_POINT
DBGEOM_CURVE
DBGEOM_LINESTRING
DBGEOM_SURFACE
DBGEOM_POLYGON
DBGEOM_COLLECTION
DBGEOM_MULTISURFACE
DBGEOM_MULTIPOLYGON
DBGEOM_MULTICURVE
DBGEOM_MULTILINESTRING
DBGEOM_MULTIPOINT

3.2 Geometry Components—Interfaces and Classes

3.2.1 Component Overview

The Geometry interfaces provide methods for accessing, analyzing, and performing operations on geometric objects. The interfaces described are based upon [1] and the detailed geometry model presented

[10] Values of OGIS_Geometry Enumerator are defined in OLEDBGIS.h, supplied separately.

in section 2.2. The instantiable geometry objects are Point, LineString, Polygon, MultiPoint, MultiLineString , MultiPolygon and GeometryCollection. Each instantiable object class must implement a number of mandatory (core) interfaces. Different geometry classes will require other interfaces specific to their properties and behavior. These other interfaces are more advanced and considered optional.

3.2.2 A Note on Inheritance

The IDL below uses *interface* inheritance sparingly. The conceptual inheritance model specified by the Geometry Object Model requires that all objects of a class C support the (mandatory) methods defined on C and on all ancestor classes of C in the class hierarchy. In COM this implies that all the interfaces defined on C and on its ancestor classes should be supported on any instance of C. For example, all LineStrings must support the `ILineString`, `ICurve` and `IGeometry` interfaces. It does not necessarily imply that the `ILineString` interface inherit from the `ICurve` interface and that `ICurve` inherit from `IGeometry`. The latter is avoided in this IDL in order to retain maximum flexibility of implementation for implementers of this specification and room for future expansion of the class hierarchy.

3.2.3 Interfaces and Classes

```
The detailed interface specification is defined using Microsoft’s Interface Definition
Language (MIDL).interface IMultiCurve;

interface IMultiPoint;


//

// Geometry interfaces

//

[ object, uuid(6A124031-FE38-11d0-BECE-00805F7C4268) ]

interface IGeometry : IUnknown

{

  [propget] HRESULT Dimension([out, retval] long * dimension);

  [propget] HRESULT SpatialReference([out, retval] ISpatialReference** spatialRef);

  [propputref] HRESULT SpatialReference([in] ISpatialReference* spatialRef);

  [propget] HRESULT IsEmpty([out, retval] VARIANT_BOOL * isEmpty);

  HRESULT SetEmpty();

  [propget] HRESULT IsSimple([out, retval] VARIANT_BOOL * isSimple);

  HRESULT Envelope([out, retval] IGeometry** envelope);

  HRESULT Clone([out, retval] IGeometry ** newShape);

  HRESULT Project([in] ISpatialReference * newSystem, [out, retval] IGeometry **result);

  HRESULT Extent2D([out] double* minX, [out] double* minY, [out] double* maxX, [out]
double* maxY);

};



[ object, uuid(6A124032-FE38-11d0-BECE-00805F7C4268) ]

interface IWks : IUnknown

{

  HRESULT ExportToWKB([in, out] VARIANT* wkb);

  HRESULT ExportToWKT([out, retval] BSTR* wkt);

  HRESULT ImportFromWKB([in] VARIANT wkb, [in] ISpatialReference* spatialRef);
  HRESULT ImportFromWKT([in] BSTR wkt, [in] ISpatialReference* spatialRef);
};

[ object, uuid(6A124033-FE38-11d0-BECE-00805F7C4268) ]
interface IGeometryFactory : IUnknown
{
  HRESULT CreateFromWKB([in] VARIANT wkb, [in] ISpatialReference* spatialRef, [out,
retval] IGeometry** geometry);
  HRESULT CreateFromWKT([in] BSTR wkt, [in] ISpatialReference* spatialRef, [out, retval]
IGeometry** geometry);
}

[ object, uuid(6A124035-FE38-11d0-BECE-00805F7C4268) ]
interface IPoint : IGeometry
{
 HRESULT Coords([out] double* x, [out] double * y);
 [propget] HRESULT X([out, retval] double * x);  // for OLE Automation Clients
 [propget] HRESULT Y([out, retval] double * y); // for Automation Clients
};

[ object, uuid(6A124036-FE38-11d0-BECE-00805F7C4268) ]
interface ICurve : IGeometry
{
  [propget] HRESULT Length([out, retval] double* value);
  HRESULT StartPoint([out, retval] IPoint** sp);
  HRESULT EndPoint([out, retval] IPoint** ep);
  [propget] HRESULT IsClosed([out, retval] VARIANT_BOOL * isClosed);
  HRESULT Value([in] double t, [out, retval] IPoint** p);
};

[ object, uuid(6A124037-FE38-11d0-BECE-00805F7C4268) ]
interface ILineString : ICurve
{
  [propget] HRESULT NumPoints  ([out, retval] long * numPoints);
  HRESULT Point ([in] long index, [out, retval] IPoint ** point);
};

[ object, uuid(6A124038-FE38-11d0-BECE-00805F7C4268) ]
interface ILinearRing : ILineString
{
};

[ object, uuid(6A124039-FE38-11d0-BECE-00805F7C4268) ]
interface ISurface : IGeometry
{
  [propget] HRESULT Area   ([out, retval] double* area);
  HRESULT Centroid ([out, retval] IPoint** result);
  HRESULT PointOnSurface  ([out, retval] IPoint** result);
};

[ object, uuid(6A12403A-FE38-11d0-BECE-00805F7C4268) ]
interface IGeometryCollection : IGeometry
{
  [propget] HRESULT NumGeometries ([out, retval] long*   numberOf);
  HRESULT   Geometry ([in] long index, [out, retval] IGeometry** geometry);
}


[ object, uuid(6A12403C-FE38-11d0-BECE-00805F7C4268) ]
interface IPolygon : ISurface
{
  HRESULT ExteriorRing([out, retval] ILinearRing ** exteriorRing);
  [propget] HRESULT NumInteriorRings([out, retval] long * count);
  HRESULT InteriorRing([in] long index, [out] ILinearRing ** interiorRing);
}

[ object, uuid(6A12403D-FE38-11d0-BECE-00805F7C4268) ]
interface IMultiCurve : IGeometryCollection
{
  [propget] HRESULT Length   ([out, retval] double* length);
  [propget] HRESULT IsClosed([out, retval] VARIANT_BOOL * isClosed);
};

[ object, uuid(6A12403F-FE38-11d0-BECE-00805F7C4268) ]
interface IMultiSurface : IGeometryCollection
{
  [propget] HRESULT Area   ([out, retval] double* area);
  HRESULT Centroid ([out, retval] IPoint** result);
  HRESULT PointOnSurface  ([out, retval] IPoint** result);
};


// implementors may choose to implement one or both of the Spatial Relation interfaces.

[ object, uuid(6A124040-FE38-11d0-BECE-00805F7C4268) ]
interface ISpatialRelation : IUnknown
{
  HRESULT Equals  ([in] IGeometry* other, [out, retval] VARIANT_BOOL * equals);
  HRESULT Touches ([in] IGeometry* other, [out, retval] VARIANT_BOOL * touches);
  HRESULT Contains([in] IGeometry* other, [out, retval] VARIANT_BOOL * contains);
  HRESULT Within  ([in] IGeometry* other, [out, retval] VARIANT_BOOL * within);
  HRESULT Disjoint([in] IGeometry* other, [out, retval] VARIANT_BOOL * disjoint);
  HRESULT Crosses ([in] IGeometry* other, [out, retval] VARIANT_BOOL * crosses);
  HRESULT Overlaps([in] IGeometry* other, [out, retval] VARIANT_BOOL * overlaps);
  HRESULT Intersects([in] IGeometry* other, [out, retval] VARIANT_BOOL * overlaps);
}

[ object, uuid(6A124041-FE38-11d0-BECE-00805F7C4268) ]
interface ISpatialRelation2 : IUnknown
{
  HRESULT Relate ([in] IGeometry* other,
                  [in] BSTR patternMatrix, [out, retval] VARIANT_BOOL* isRelated);
};

// The ISpatialOperator interface groups the commonly used and agreed upon operators
// into a single interface. Operators that are less commonly used can be moved into a
// separate ISpatialOperator2 interface.
// The goal has been to minimize the number of interfaces that clients deal with.

[ object, uuid(6A124042-FE38-11d0-BECE-00805F7C4268) ]
interface ISpatialOperator : IUnknown
{
  // Proximity Operators

  HRESULT Distance([in] IGeometry* other, [out, retval] double* distance);

  // Topological Operators

  HRESULT Boundary([out, retval] IGeometry ** boundary);

  // Analysis - Constructive Operators

  HRESULT Buffer([in] double distance, [out, retval] IGeometry** result);
  HRESULT ConvexHull([out, retval] IGeometry ** result);

  // Set Theoretic Operators

  HRESULT Intersection ([in] IGeometry* other, [out] IGeometry ** result);
  HRESULT Union([in] IGeometry* other,  [out] IGeometry ** result);
  HRESULT Difference([in] IGeometry* other, [out] IGeometry ** result);
  HRESULT SymmetricDifference([in] IGeometry* other, [out] IGeometry ** result);

};
```

```
#if defined(_GEOMETRY_)

// example of a specific geometry library implementation

[   uuid(6A124045-FE38-11d0-BECE-00805F7C4268),
    lcid(0x0000),
    version(1.0)
]

library Geometry
{

// COM Classes would be defined here. CoTypes are shown instead :

/*************************************************************************
    coType Point
    {
        [mandatory] interface IGeometry;
        [mandatory] interface IPoint;
        [mandatory] interface IWks;

        [optional]  interface ISpatialRelation;
        [optional]  interface ISpatialRelation2;
        [optional]  interface ISpatialOperator;

    };

    coType LineString
    {
        [mandatory] interface IGeometry;
        [mandatory] interface ICurve;
        [mandatory] interface ILineString;
        [mandatory] interface IWks;

        [optional]  interface ISpatialRelation;
        [optional]  interface ISpatialRelation2;
        [optional]  interface ISpatialOperator;

    };

    coType Polygon
```

```
{
    [mandatory] interface IGeometry;
    [mandatory] interface ISurface;
    [mandatory] interface IPolygon;
    [mandatory] interface IWks;

    [optional]  interface ISpatialRelation;
    [optional]  interface ISpatialRelation2;
    [optional]  interface ISpatialOperator;
};


coType MultiPoint
{
    [mandatory] interface IGeometry;
    [mandatory] interface IGeometryCollection;
    [mandatory] interface IWks;

    [optional]  interface ISpatialRelation;
    [optional]  interface ISpatialRelation2;
    [optional]  interface ISpatialOperator;
};


coType MultiLineString
{
    [mandatory] interface IGeometry;
    [mandatory] interface IGeometryCollection;
    [mandatory] interface IMultiCurve;
    [mandatory] interface IWks;

    [optional]  interface ISpatialRelation;
    [optional]  interface ISpatialRelation2;
    [optional]  interface ISpatialOperator;
};



coType MultiPolygon
{
    [mandatory] interface IGeometry;
    [mandatory] interface IGeometryCollection;
    [mandatory] interface IMultiSurface;
    [mandatory] interface IWks;
```

```
        [optional]  interface ISpatialRelation;
        [optional]  interface ISpatialRelation2;
        [optional]  interface ISpatialOperator;
    };

    coType GeometryCollection
    {
        [mandatory] interface IGeometry;
        [mandatory] interface IGeometryCollection;
        [mandatory] interface IWks;

        [optional]  interface ISpatialRelation;
        [optional]  interface ISpatialRelation2;
        [optional]  interface ISpatialOperator;
    };

    coType GeometryFactory
    {
        [mandatory] interface IGeometryFactory;
    }

************************************************************/
}

#endif
```

3.2.4 Description

This section provides a technical narrative describing the meaning, structure, and behavior of the interfaces comprising the Geometry proposal.

3.2.4.1 IGeometry Interface

The `IGeometry` interface defines the common basic properties and methods for all geometric objects.

3.2.4.1.1 IGeometry::get_Dimension

```
HRESULT get_Dimension(long * dimension);
```

Returns the dimension of the geometry, 0 for Points and MultiPoints, 1 for Curves and MultiCurves, 2 for Surfaces and MultiSurfaces.

Parameters

`dimension [out]`—The dimension of the geometry.

Return Code

S_OK—The method succeeded

3.2.4.1.2 IGeometry::get_SpatialReference

```
HRESULT get_SpatialReference(ISpatialReference ** spatialRef);
```

Returns the spatial reference system associated with the geometry. A geometry may not have had a spatial reference system defined for it, in which case *spatialRef will be NULL on exit.

Parameters

spatialRef [out]—The spatial reference system associated with the shape.

Return Code

S_OK—The method succeeded

3.2.4.1.3 IGeometry::put_SpatialReference

```
HRESULT put_SpatialReference(ISpatialReference * spatialRef);
```

Sets the spatial reference system of the geometry. If no spatial reference is currently assigned to the geometry, then the effect of this method is trivial. If a spatial reference was already defined, this method causes the geometry to be projected into the new spatial reference system.

Parameters

spatialRef [in]—The new spatial reference system for the shape. May be NULL.

Return Code

S_OK—The method succeeded

3.2.4.1.4 IGeometry::IsEmpty

```
HRESULT IsEmpty(VARIANT_BOOL* isEmpty);
```

Tests if a geometry is empty.

Parameters

isEmpty [out]–TRUE if the geometry is empty.

Return Code

S_OK—The method succeeded

3.2.4.1.5 IGeometry::SetEmpty

```
HRESULT SetEmpty();
```

Changes the geometry to be Empty.

Parameters

None

Return Code

`S_OK`—The method succeeded

3.2.4.1.6 IGeometry::IsSimple

```
HRESULT IsSimple(VARIANT_BOOL* isEmpty);
```

Tests if a geometry is simple. The definition of Simple for each type of Geometry is specified as part of its class description.

Parameters

`isSimple [out]`–`TRUE` if the geometry is simple.

Return Code

`S_OK`—The method succeeded

3.2.4.1.7 IGeometry::get_Envelope

```
HRESULT get_Envelope(IGeometry ** envelope);
```

Returns the bounding envelope for the geometry. The sides of the envelope are parallel to the sides of the coordinate system of the spatial reference for the geometry.

Parameters

`envelope [out]`–The bounding envelope of the geometry.

Return Code

`S_OK`—The method succeeded

3.2.4.1.8 IGeometry::Clone

```
HRESULT Clone(IGeometry ** newGeometry);
```

Create and return a new geometric object identical to the source geometry object.

Parameters

`newGeometry [out]`–A reference to a new geometry created by this method.

Return Code

`S_OK`—The method succeeded

3.2.4.1.9 IGeometry::Project

```
HRESULT Project(ISpatialReferenceSystem * newSystem, IGeometry ** result);
```

Return a new geometry object that is the result of projection applied to the source geometry. If the source is not in a geographic coordinate system, the inverse projection of its current coordinate system is applied first. The operation can cause a loss of data. The source geometry may be clipped by some line on the earth's surface to insure that all points in result are in the range of the projection.

Projecting a geometry is not a method on a SpatialReferenceSystem because projecting a Geometry to maximize the correctness of the resulting geometry requires knowledge of the structure of the Geometry being projected. The SpatialReferenceSystem provides functionality to project individual coordinates, each Geometry class uses this primitive as appropriate. This design is also extensible with the addition of new Geometry classes.

Parameters

`newSystem[in]`–A definition of the coordinate system to which the source geometry will be projected.

`result [out]`–A new geometry containing the projected version of the source geometry.

Return Code

`S_OK`—The method succeeded

3.2.4.2 IPoint Interface

The `IPoint` interface defines methods for accessing the coordinates and measure of a point object.

3.2.4.2.1 IPoint::get_X

```
HRESULT get_X(double * x);
```

Returns the x-coordinate.

Parameters

`x [out]`–The x-coordinate.

Return Code

`S_OK`—The method succeeded

3.2.4.2.2 IPoint::get_Y

```
HRESULT get_Y(double * y);
```

Returns the y-coordinate.

Parameters

`y [out]`–The y-coordinate.

Return Code

S_OK—The method succeeded

3.2.4.2.3 IPoint::get_Coords

```
HRESULT get_Coords (double * x, double * y);
```

Gets the x, y-coordinates of the point

Parameters

x, y [out]–The x, y-coordinates.

Return Code

S_OK—The method succeeded

3.2.4.3 ICurve Interface

The ICurve interface defines methods for accessing the general properties of curves as defined in the geometry specification.

3.2.4.3.1 ICurve::get_Length

```
HRESULT get_Length(double * value);
```

Returns the length of the curve.

Parameters

value [out]–The length of the curve.

Return Code

S_OK—The method succeeded

3.2.4.3.2 ICurve::get_StartPoint

```
HRESULT get_StartPoint(IPoint ** startPoint);
```

Returns a copy of the starting point of the curve.

Parameters

startPoint [out]–The starting point of the curve.

Return Code

S_OK—The method succeeded

3.2.4.3.3 ICurve::get_EndPoint

```
HRESULT get_EndPoint(IPoint ** endPoint);
```

Returns a copy of the end point of the curve.

Parameters

`endPoint [out]`–The end point of the curve.

Return Code

`S_OK`—The method succeeded

3.2.4.3.4 ICurve::IsClosed

```
HRESULT IsClosed(VARIANT_BOOL* isEmpty);
```

Tests if a curve is closed. A curve is closed if its start point is equal to its end point.

Parameters

`isClosed [out]`–`TRUE` if the curve is closed.

Return Code

`S_OK`—The method succeeded

3.2.4.3.5 ICurve::get_Value

```
HRESULT get_Value(double t, IPoint ** p);
```

Returns the point p at distance t ***along the curve*** from the start point. `value(0.0) = startPoint`, `value(Length) = endPoint`.

Parameters

`t [in]`—The distance along the curve.

`p [out]`–The point on the curve.

Return Code

`S_OK`—The method succeeded

3.2.4.4 ILineString Interface

The `ILineString` interface defines methods for accessing the properties of linestrings as defined in the geometry specification.

3.2.4.4.1 ILineString::get_NumPoints

```
HRESULT get_NumPoints(long * numPoints);
```

Returns the number of points in the linestring.

Parameters

`numPoints [out]`–The number of points in the linestring.

Return Code

`S_OK`—The method succeeded

3.2.4.4.2 ILineString::get_Point

```
HRESULT get_Point(long index, IPoint ** result);
```

Returns the point at the specified index. The range of the index is from `0` to `NumPoints-1`.

Parameters

`index [in]`–The index of the point to return.

`result [out]`–An interface on a geometric point object that is a copy of the specified point of the linestring.

Return Code

`S_OK`—The method succeeded

3.2.4.5 ISurface interface

This interface describes operations that are available on all surfaces.

3.2.4.5.1 ISurface::get_Area

```
HRESULT get_Area(double * area);
```

Returns the total area of the surface geometry.

Parameters

`area [out]`–The area of the geometry.

Return Code

`S_OK`—The method succeeded

3.2.4.5.2 ISurface::get_Centroid

```
HRESULT get_Centroid(IPoint ** center);
```

Returns the centroid of the geometry. The centroid point is not necessarily contained within the geometry.

Parameters

`result [out]`–The center of the geometry.

Return Code

`S_OK`—The method succeeded

3.2.4.5.3 ISurface::get_PointOnSurface

```
HRESULT get_PointOnSurface(IPoint ** label);
```

Returns a point guaranteed to be on the surface (Within the geometry).

Parameters

`result [out]`–a point on the surface.

Return Code

`S_OK`—The method succeeded

3.2.4.6 IPolygon interface

The `IPolygon` interface allows a client to obtain information about the rings in a Polygon.

3.2.4.6.1 IPolygon::get_ExteriorRing

```
HRESULT get_ExteriorRing(ILinearRing **ering);
```

Returns the exterior ring of the source polygon..

Parameters

`ering [out]`–The exterior ring for the polygon.

Return Code

`S_OK`—The method succeeded

3.2.4.6.2 IPolygon::get_NumInteriorRings

```
HRESULT get_NumInteriorRings(long* count) ;
```

Return the number of interior rings in the source polygon..

Parameters

`count [out]`–The number of interior Rings for the polygon.

Return Code

`S_OK`—The method succeeded

3.2.4.6.3 IPolygon::get_InteriorRing

```
HRESULT get_InteriorRing(long index, ILinearRing** interiorRing) ;
```

Return the specified interior ring for the source polygon. The index order of rings is assigned by the Polygon and is not guaranteed to have any geometric significance. Index range is `0` to `NumRings-1`.

Parameters

`index [in]`—The position of the ring to retrieve

`interiorRing [out]`–The specified `interiorRing` for the source polygon.

Return Code

`S_OK`—The method succeeded

3.2.4.7 IGeometryCollection Interface

The `IGeometryCollection` interface is supported by geometry collection objects. For purposes of element access it supports a view of the elements of the collection as an indexed set. The order of the elements in the set may be one that is imposed by the collection and is not guaranteed to have any geometric significance.

3.2.4.7.1 IGeometryCollection::get_NumGeometries

```
HRESULT get_NumGeometries(long* numGeometries);
```

Returns the number of element geometries in the geometry collection.

Parameters

`numGeometries [out]`–The number of elements.

Return Code

`S_OK`—The method succeeded

3.2.4.7.2 IGeometryCollection::get_Geometry

```
HRESULT get_Geometry(long index, IGeometry ** geometry);
```

Returns the geometry at the specified index in the GeometryCollection. The consumer of this interface receives a direct reference on the specified element of the collection and no copy is made. Index range is `0` to `NumGeometries-1`.

Parameters

`index [in]`–The index of the element to return.

`geometry [out]`—the geometry at the specified index.

Return Code

`S_OK`—The method succeeded

3.2.4.8 IMultiSurface interface

This interface describes operations that are available on all multi surfaces.

3.2.4.8.1 IMultiSurface::get_Area

```
HRESULT get_Area(double * area);
```

Returns the total area of the geometry.

Parameters

`area [out]`–The area of the geometry.

Return Code

`S_OK`—The method succeeded

3.2.4.8.2 IMultiSurface::get_Centroid

```
HRESULT get_Centroid(IPoint ** center);
```

Returns the centroid of the geometry. This point is not necessarily contained within the geometry.

Parameters

`result [out]`–The center of the geometry.

Return Code

`S_OK`—The method succeeded

3.2.4.8.3 IMultiSurface::get_PointOnSurface

```
HRESULT get_PointOnSurface(IPoint ** label);
```

Returns a point guaranteed to be on the MultiSurface (Within the geometry).

Parameters

`result [out]`–a point on the surface.

Return Code

`S_OK`—The method succeeded

3.2.4.9 IMultiCurve Interface

The `IMultiCurve` interface defines methods common to all MultiCurves.

3.2.4.9.1 IMultiCurve::get_Length

```
HRESULT get_Length(double * value);
```

Returns the length of the MultiCurve.

Parameters

`value [out]`–The length of the MultiCurve.

Return Code

`S_OK`—The method succeeded

3.2.4.9.2 IMultiCurve::IsClosed

```
HRESULT IsClosed(VARIANT_BOOL* isEmpty);
```

Tests if the multicurve is closed.

Parameters

`isClosed [out]`–`TRUE` if the multi curve is closed.

Return Code

`S_OK`—The method succeeded

3.2.4.10 ISpatialRelation Interface

The `ISpatialRelation` interface defines a set of named spatial relationship operators for geometric shape objects. The behavior of these operators is described in detail in the geometry object model sub-section of the Architecture section of this specification.

3.2.4.10.1 ISpatialRelation::Equals

```
HRESULT Equals(IGeometry * otherGeometry, VARIANT_BOOL* result);
```

Returns `TRUE` if `otherGeometry` is of the same type and defines the same point set as the source geometry.

Parameters

`otherGeometry [in]`–The comparison geometry.

`result [out]`–The result of the test.

Return Code

`S_OK`—The method succeeded

3.2.4.10.2 ISpatialRelation::Touches

```
HRESULT Touches(IGeometry * otherGeometry, VARIANT_BOOL* result);
```

Returns `TRUE` if the only points in common between the two geometries lie in the union of their boundaries.

Parameters

otherGeometry [in]–The comparison geometry.

result [out]–The result of the test.

Return Code

S_OK—The method succeeded

3.2.4.10.3 ISpatialRelation::Contains

```
HRESULT Contains(IGeometry * otherGeometry, VARIANT_BOOL* result);
```

Returns TRUE if otherGeometry is wholly contained within the source geometry. This is the same as reversing the primary and comparison shapes of the Within operation

Parameters

otherGeometry [in]–The comparison shape.

result [out]–The result of the test.

Return Code

S_OK—The method succeeded

3.2.4.10.4 ISpatialRelation::Within

```
HRESULT Within(IGeometry * otherGeometry, VARIANT_BOOL* result);
```

Returns TRUE if the primary geometry is wholly contained within the comparison geometry.

Parameters

otherGeometry [in]–The comparison shape.

result [out]–The result of the test.

Return Code

S_OK—The method succeeded

3.2.4.10.5 ISpatialRelation::Disjoint

```
HRESULT Disjoint(IGeometry * otherGeometry, VARIANT_BOOL* result);
```

Returns TRUE if otherGeometry is disjoint from the source geometry.

Parameters

otherGeometry [in]–The comparison geometry.

`result [out]`–The result of the test.

Return Code

`S_OK`—The method succeeded

3.2.4.10.6 ISpatialRelation::Crosses

```
HRESULT Crosses(IGeometry * otherGeometry, VARIANT_BOOL* result);
```

Returns `TRUE` if the intersection of the two geometries results in a geometry whose dimension is less than the maximum dimension of the two geometries and the intersection geometry is not equal to either geometry.

Parameters

`otherGeometry [in]`–The comparison shape.

`result [out]`–The result of the test.

Return Code

`S_OK`—The method succeeded

3.2.4.10.7 ISpatialRelation::Overlaps

```
HRESULT Overlaps(IGeometry * otherGeometry, VARIANT_BOOL* result);
```

Returns `TRUE` if the intersection of the two geometries results in an object of the same dimension as the input geometries and the intersection geometry is not equal to either geometry.

Parameters

`otherGeometry [in]`–The comparison shape.

`result [out]`–The result of the test.

Return Code

`S_OK`—The method succeeded

3.2.4.10.8 ISpatialRelation::Intersects

```
HRESULT Intersects(IGeometry * otherGeometry, VARIANT_BOOL* result);
```

Returns `TRUE` if there is any intersection between the two geometries.

Parameters

`otherGeometry [in]`–The comparison shape.

`result [out]`–The result of the test.

Return Code

S_OK—The method succeeded

3.2.4.11 ISpatialRelation2 Interface

The ISpatialRelation2 interface defines a set of lower level building block spatial relationship operators for geometric objects. These operators are described fully in the accompanying geometry model specification

3.2.4.11.1 ISpatialRelation2::Relate

```
HRESULT Relate(IGeometry* anotherGeometry, char* testPatternMatrix,
VARIANT_BOOL* result);
```

Determines if the spatial relationship specfied by the testPatternMatrix defined over the DE-9IM holds TRUE. For a definition of the testPatternMatrix see the Geometry Model Document.

Parameters

anotherGeometry [in]–The geometry to test against.

testPatternMatrix [in]—The test pattern matrix.

result [out]–The result of the test..

Return Code

S_OK—The method succeeded

3.2.4.12 ISpatialOperator Interface

The ISpatialOperator interface packages a number of common Proxmity, Constructive, Topological and Set operators on geometries.

3.2.4.12.1 ISpatialOperator::Boundary

```
HRESULT Boundary(IGeometry ** outGeometry);
```

Returns the closure of the combinatorial boundary of the source Geometry in outGeometry. For definitions see [1]. and section 2.2.

Parameters

oputGeometry [out]–The closure of the combinatorial boundary of the source geometry.

Return Code

S_OK—The method succeeded

3.2.4.12.2 ISpatialOperator::Distance

```
HRESULT Distance(IGeometry * otherGeometry, double * distance);
```

Returns the minimum distance between the source geometry and the `otherGeometry`. The distance units are in terms of the spatial reference system associated with the source geometry. For example, this method returns zero when its operands are two intersecting polylines, regardless of the relationships of the polylines' vertices.

Parameters

`otherGeometry [in]`–The comparison geometry.

`distance [out]`–The distance between the shapes.

Return Code

`S_OK`—The method succeeded

3.2.4.12.3 ISpatialOperator::Buffer

```
HRESULT Buffer(double distance, IGeometry ** result);
```

Returns a polygon that includes all points within `distance` units of the source geometry's boundary.

Parameters

`distance [in]`–The buffer distance around the shape.

`result [out]`–The resulting polygon.

Return Code

`S_OK`—The method succeeded

3.2.4.12.4 ISpatialOperator::Intersection

```
HRESULT Intersection(IGeometry * otherGeometry, IGeometry ** result);
```

Returns a geometry that represents the intersection of the source geometry with `otherGeometry`.

Parameters

`otherGeometry [in]`–The other operand to the intersection operation.

`result [out]`–The resulting geometry.

Return Code

`S_OK`—The method succeeded

3.2.4.12.5 ISpatialOperator::Union

```
HRESULT Union(IGeometry * otherGeometry, IGeometry ** result);
```

Returns a geometry that represents the union of the source geometry with `otherGeometry`.

Parameters

`otherGeometry [in]`–The comparison geometry.

`result [out]`–The resulting geometry.

Return Code

`S_OK`—The method succeeded

3.2.4.12.6 ISpatialOperator::Difference

```
HRESULT Difference(IGeometry * otherGeometry, IGeometry ** result);
```

Returns a geometry that represents the difference between the source geometry and `otherGeometry` (`<source geometry> - otherGeometry`).

Parameters

`other [in]`–The comparison geometry collections.

`result [out]`–The resulting geometry.

Return Code

`S_OK`—The method succeeded

3.2.4.12.7 ISpatialOperator::SymmetricDifference

```
HRESULT SymmetricDifference(IGeometry * otherGeometry, IGeometry ** result);
```

Returns a geometry that represents the symmetric difference between the source geometry and `otherGeometry`.

Parameters

`other [in]`–The comparison geometry.

`result [out]`–The resulting geometry.

Return Code

`S_OK`—The method succeeded

3.2.4.12.8 ISpatialOperator::ConvexHull

```
HRESULT ConvexHull(IGeometry ** result);
```

Returns the convex hull of the source geometry.

Parameters

`result [out]`–The resulting geometry. For non-trivial cases this will be a polygon. However, the convex hull of a Point should be a Point. Similarly, the convex hull of a straight LineString may be a LineString.

Return Code

S_OK—The method succeeded

3.2.4.13 IWks Interface

The IWks interface defines methods for dealing with Well-known external representations of Geometry. The external reprentation(s) supported are the Well-known Binary Representation for Geometry (WKBGeometry) described in this proposal.

3.2.4.13.1 IWks::ExportToWKB

```
HRESULT ExportToWKB(VARIANT * buffer);
```

Copies the Well-known binary representation of the geometry. The type of the variant will be an array of unsigned bytes (VT_ARRAY|VT_UI1).

Parameters

buffer [in, out]–The buffer that will hold the Well-known binary representation for the geometry.

Return Code

S_OK—The method succeeded

3.2.4.13.2 IWks::ExportToWKT

```
HRESULT ExportToWKT(BSTR * wkt);
```

Returns the Well-known text representation of the geometry.

Parameters

wkt [out, retval]–The returned Well-known text representation for the geometry.

Return Code

S_OK—The method succeeded

3.2.4.13.3 IWks::ImportFromWKB

```
HRESULT ImportFromWKB(VARIANT buffer, ISpatialReference * spatialRef);
```

Imports the input Well-known Binary Representation into the object. This method may be used to initialize a geometry object after creating it using the generic COM ClassFactory for its CoClass. The type of object stored in the input WKBGeometry must match the class of the source object.

Parameters

buffer [in]–The buffer containing the Well-known binary representation for geometry (WKBGeometry).

spatialRef [in]–The spatial reference system of the Well-known binary.

Return Code

S_OK—The method succeeded

3.2.4.13.4 IWks::ImportFromWKT

```
HRESULT ImportFromWKT(BSTR buffer, ISpatialReference * spatialRef);
```

Imports the input Well-known Text Representation into the object. This method may be used to initialize a geometry object after creating it using the generic COM ClassFactory for its CoClass. The type of object stored in the input WKBGeometry must match the class of the source object.

Parameters

buffer [in]–The buffer containing the Well-known text representation for geometry.

spatialRef [in]–The spatial reference system of the Well-known text.

Return Code

S_OK—The method succeeded

3.2.4.14 IGeometryFactory Interface

The IGeometryFactory interface defines methods for creating a geometry given its Well-known Representation and does not require the client to know the type of geometry represented. It is a convenient mechanism for creating a set of geometry objects, belonging to different classes, given their Well-known representations.

3.2.4.14.1 IGeometryFactory::CreateFromWKB

```
HRESULT CreateFromWKB(VARIANT * buffer, ISpatialReference*
spatialRef,IGeometry** result);
```

Creates a Geometry object from the input Well-known Binary representation for Geometry (WKBGeometry).

Parameters

buffer [in]–The buffer that holds the Well-known binary representation for geometry.

spatialRef [in]–The spatial reference of the Well-known binary.

result [out]–The output geometry.

Return Code

S_OK—The method succeeded

3.2.4.14.2 IGeometryFactory::CreateFromWKT

```
HRESULT CreateFromWKT(BSTR buffer, ISpatialReference* spatialRef,IGeometry**
result);
```

Creates a Geometry object from the input Well-known Text representation for Geometry (`WKBGeometry`).

Parameters

`buffer [in]`–The buffer that holds the Well-known text representation for geometry.

`spatialRef [in]`–The spatial reference of the Well-known text.

`result [out]`–The output geometry.

Return Code

`S_OK`—The method succeeded

3.2.5 Exceptions, Errors, and Error Codes

The `HRESULT`s of the methods of the interfaces are used to indicate errors. A return value of `S_OK` indicates successful completion, any other return value implies a provider specific error occurred.

3.3 The Well-known Binary Representation for Geometry (WKBGeometry)

3.3.1 Component Overview

The Well-known Binary Representation for Geometry (`WKBGeometry`) provides a portable representation of a Geometry value as a contiguous stream of bytes. It permits Geometry values to be exchanged between an ODBC client and an SQL database in binary form.

3.3.2 Component Description

The Well-known Binary Representation for Geometry is obtained by serializing a geometry instance as a sequence of numeric types drawn from the set {`Unsigned Integer`, `Double`} and then serializing each numeric type as a sequence of bytes using one of two well defined, standard, binary representations for numeric types (NDR, XDR). The specific binary encoding (NDR or XDR) used for a geometry byte stream is described by a one byte tag that precedes the serialized bytes. The only difference between the two encodings of geometry is one of byte order, the XDR encoding is Big Endian, the NDR encoding is Little Endian.

3.3.2.1 Numeric Type Definitions

An `Unsigned Integer` is a 32-bit (4-byte) data type that encodes a nonnegative integer in the range [0, 4294967295].

A `Double` is a 64-bit (8-byte) double precision data type that encodes a double precision number using the IEEE 754 double precision format

The above definitions are common to both XDR and NDR.

3.3.2.2 XDR (Big Endian) Encoding of Numeric Types

The XDR representation of an `Unsigned Integer` is Big Endian (most significant byte first).

The XDR representation of a `Double` is Big Endian (sign bit is first byte).

3.3.2.3 NDR (Little Endian) Encoding of Numeric Types

The NDR representation of an `Unsigned Integer` is Little Endian (least significant byte first).

The NDR representation of a `Double` is Little Endian (sign bit is last byte).

3.3.2.4 Conversion between the NDR and XDR representations of WKBGeometry

Conversion between the NDR and XDR data types for `Unsigned Integer` and `Double` numbers is a simple operation involving reversing the order of bytes within each `Unsigned Integer` or `Double` number in the byte stream.

3.3.2.5 Relationship to other COM and CORBA data transfer protocols

The XDR representation for `Unsigned Integer` and `Double` numbers described above is also the standard representation for `Unsigned Integer` and for `Double` number in the CORBA Standard Stream Format for Externalized Object Data that is described as part of the CORBA Externalization Service Specification [15].

The NDR representation for `Unsigned Integer` and `Double` number described above is also the standard representation for `Unsigned Integer` and for `Double` number in the DCOM protocols that is based on DCE RPC and NDR [16].

3.3.2.6 Description of WKBGeometry Byte Streams

The Well-known Binary Representation for Geometry is described below. The basic building block is the byte stream for a `Point`, which consists of two `Double` numbers. The byte streams for other geometries are built using the byte streams for geometries that have already been defined.

```
// Basic Type definitions
// byte : 1 byte
// uint32 : 32 bit unsigned integer (4 bytes)
// double : double precision number (8 bytes)

// Building Blocks : Point, LinearRing

Point {
           double x;
           double y;
};

LinearRing  {
           uint32  numPoints;
           Point   points[numPoints];
}

enum wkbGeometryType {
           wkbPoint = 1,
           wkbLineString = 2,
           wkbPolygon = 3,
           wkbMultiPoint = 4,
           wkbMultiLineString = 5,
           wkbMultiPolygon = 6,
           wkbGeometryCollection = 7
};

enum wkbByteOrder {
           wkbXDR = 0,            // Big Endian
           wkbNDR = 1             // Little Endian
};

WKBPoint {
           byte           byteOrder;
           uint32         wkbType;                          // 1
           Point          point;
}

WKBLineString {
           byte           byteOrder;
           uint32         wkbType;                          // 2
           uint32         numPoints;
           Point          points[numPoints];
}

WKBPolygon {
           byte           byteOrder;
           uint32         wkbType;                          // 3
           uint32         numRings;
           LinearRing     rings[numRings];
}

WKBMultiPoint {
           byte           byteOrder;
           uint32         wkbType;                  // 4
           uint32         num_wkbPoints;
           WKBPoint       WKBPoints[num_wkbPoints];
}

WKBMultiLineString {
           byte           byteOrder;
           uint32         wkbType;                          // 5
           uint32         num_wkbLineStrings;
           WKBLineString  WKBLineStrings[num_wkbLineStrings];
}

wkbMultiPolygon {
           byte           byteOrder;
           uint32         wkbType;                          // 6
           uint32         num_wkbPolygons;
           WKBPolygon     wkbPolygons[num_wkbPolygons];
}

WKBGeometry {
           union {
               WKBPoint                  point;
               WKBLineString             linestring;
               WKBPolygon                polygon;
               WKBGeometryCollection     collection;
               WKBMultiPoint             mpoint;
               WKBMultiLineString        mlinestring;
               WKBMultiPolygon           mpolygon;
           }
};

WKBGeometryCollection {
           byte           byte_order;
           uint32         wkbType;                                  // 7
           uint32         num_wkbGeometries;
           WKBGeometry    wkbGeometries[num_wkbGeometries];
}
```

Figure 3.2 shows a pictorial representation of the Well-known Byte Stream for a `Polygon` with one outer ring and one inner ring.

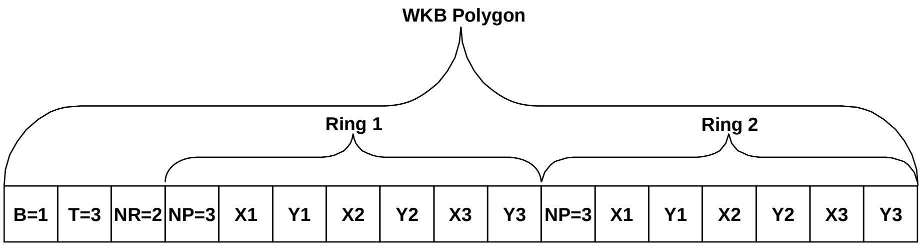


Figure 3.2—Well-known Binary Representation for a `Geometry` value in NDR format (B=1) of type Polygon (T=3) with 2 linear rings (NR = 2) each ring having 3 points (NP = 3).

3.3.2.7 Assertions for Well-known Binary Representation for Geometry

The Well-known Binary Representation for Geometry is designed to represent instances of the geometry types described in the Geometry Object Model and in the OpenGIS Abstract Specification. **Any `WKBGeometry` instance must satisfy the assertions for the type of `Geometry` that it describes**. These assertions may be found in the section 2.2.

These assertions imply the following for Rings, Polygons and MultiPolygons:

3.3.2.8 Linear Rings

Rings are simple and closed, which means that Linear Rings may **not** self-touch.

3.3.2.9 Polygons

No two Linear Rings in the boundary of a Polygon may cross each other, the Linear Rings in the boundary of a polygon may intersect at most at a single point but only as a tangent.

3.3.2.10 MultiPolygons

1. The interiors of 2 Polygons that are elements of a MultiPolygon may not intersect.

2. The Boundaries of any 2 Polygons that are elements of a MultiPolygon may touch at only a *finite* number of points.

For more details on the above assertions and for the assertions for each geometry type the reader is referred to the Geometry Object Model section of this specification.

3.4 Spatial Reference System Components—Interfaces and Classes

3.4.1 Component Overview

The Spatial Reference System component describes interfaces that define and describe the spatial reference system for geographic features. The Spatial Reference System interfaces allow different spatial reference systems to be defined and queried. Each geometric object will have a spatial reference system associated with it. The simplest way to define a new SpatialReference object is to provide a standard identification code to a Spatial Reference Factory to specify a predefined reference system. A custom SpatialReference object can also be defined by creating its constituent parts and setting the properties. Almost all of the spatial reference interfaces consist of access methods for object properties.

3.4.2 Interface(s), Data Structures, Language Constructs

```
import "ocidl.idl";


typedef struct WKSPoint
{
    double x;
    double y;
} WKSPoint;


[
  object, uuid(bcca38a0-fe1c-11d0-ad87-080009b6f22b)
]
interface ISpatialReferenceInfo : IUnknown
{
  [propget] HRESULT Name([out, retval] BSTR* name);
  [propput] HRESULT Name([in] BSTR name);
  [propget] HRESULT Authority([out, retval] BSTR* name);
  [propput] HRESULT Authority([in] BSTR name);
  [propget] HRESULT Code([out, retval] long* code);
  [propput] HRESULT Code([in] long code);
  [propget] HRESULT Alias([out, retval] BSTR* alias);
  [propput] HRESULT Alias([in] BSTR alias);
  [propget] HRESULT Abbreviation([out, retval] BSTR* abbrev);
  [propput] HRESULT Abbreviation([in] BSTR abbrev);
  [propget] HRESULT Remarks([out, retval] BSTR* remarks);
  [propput] HRESULT Remarks([in] BSTR remarks);
  [propget] HRESULT WellKnownText ([out, retval] BSTR* wkt);
};

[
 object, uuid(221733b0-fe1d-11d0-ad87-080009b6f22b)
]
interface IUnit : ISpatialReferenceInfo
{
};

[
 object, uuid(4febc550-fe1d-11d0-ad87-080009b6f22b)
]
interface IAngularUnit : IUnit
{
  [propget] HRESULT RadiansPerUnit([out, retval] double* factor);
  [propput] HRESULT RadiansPerUnit([in] double factor);
};

[
 object, uuid(80855df0-fe1d-11d0-ad87-080009b6f22b)
]
interface ILinearUnit : IUnit
{
  [propget] HRESULT MetersPerUnit([out, retval] double* factor);
  [propput] HRESULT MetersPerUnit([in] double factor);
};

[
 object, uuid(ce7266c0-fe1d-11d0-ad87-080009b6f22b)
]
interface IEllipsoid : ISpatialReferenceInfo
{
  [propget] HRESULT SemiMajorAxis([out, retval] double* axis);
  [propput] HRESULT SemiMajorAxis([in] double axis);
  [propget] HRESULT SemiMinorAxis([out, retval] double* axis);
  [propput] HRESULT SemiMinorAxis([in] double axis);
  [propget] HRESULT InverseFlattening([out, retval] double* invFlat);
  [propput] HRESULT InverseFlattening([in] double invFlat);
  [propget] HRESULT AxisUnit([out, retval] ILinearUnit** unit);
  [propput] HRESULT AxisUnit([in] ILinearUnit* unit);
};

[
 object, uuid(f699c510-fe1d-11d0-ad87-080009b6f22b)
]
interface IHorizontalDatum : ISpatialReferenceInfo
{
  [propget] HRESULT Ellipsoid([out, retval] IEllipsoid** ellipsoid);
  [propput] HRESULT Ellipsoid([in] IEllipsoid* ellipsoid);
};

[
 object, uuid(15129940-fe1e-11d0-ad87-080009b6f22b)
]
interface IPrimeMeridian : ISpatialReferenceInfo
{
  [propget] HRESULT Longitude([out, retval] double* longitude);
  [propput] HRESULT Longitude([in] double longitude);
  [propget] HRESULT AngularUnit([out, retval] IAngularUnit** unit);
  [propput] HRESULT AngularUnit([in] IAngularUnit* unit);
};

[
 object, uuid(4c4c5c00-fe1e-11d0-ad87-080009b6f22b)
]
interface ISpatialReference : ISpatialReferenceInfo
{
};

[
 object, uuid(7c3c56d0-fe1e-11d0-ad87-080009b6f22b)
]
interface IGeodeticSpatialReference : ISpatialReference
{
};

[
 object, uuid(a3fd5390-fe1e-11d0-ad87-080009b6f22b)
```

```
]
interface IGeographicCoordinateSystem : IGeodeticSpatialReference
{
  [propget] HRESULT Usage([out, retval] BSTR* usage);
  [propput] HRESULT Usage([in] BSTR usage);
  [propget] HRESULT HorizontalDatum([out, retval] IHorizontalDatum** datum);
  [propput] HRESULT HorizontalDatum([in] IHorizontalDatum* datum);
  [propget] HRESULT AngularUnit([out, retval] IAngularUnit** unit);
  [propput] HRESULT AngularUnit([in] IAngularUnit* unit);
  [propget] HRESULT PrimeMeridian([out, retval] IPrimeMeridian** prmMerid);
  [propput] HRESULT PrimeMeridian([in] IPrimeMeridian* prmMerid);
};

[
 object, uuid(9a5e32d0-fe1f-11d0-ad87-080009b6f22b)
]
interface IParameter : ISpatialReferenceInfo
{
  [propput] HRESULT ValueUnit([in] IUnit* unit );
  [propget] HRESULT ValueUnit([out, retval] IUnit** unit);
  [propput] HRESULT Value ([in] double value);
  [propget] HRESULT Value ([out, retval] double* value);
};

[
 object, uuid(7309b460-fe1f-11d0-ad87-080009b6f22b)
]
interface IParameterInfo : IUnknown
{
  [propget] HRESULT NumParameters ([out, retval] long* numParameters);
  [propget] HRESULT DefaultParameters ([in] long size,
                                       [out, size_is(size)] IParameter* parameters[]);
  [propget] HRESULT Parameters ([in] long size,
                                [out, size_is(size)] IParameter* parameters[]);
  [propput] HRESULT Parameters ([in] long size,
                                [in, size_is(size)] IParameter* parameters[]);
};

// subclasses of IParameterInfo may provide projection specific methods
// with type safe signatures for getting and setting parameters
// for eg.
// interface ITransverseMercatorParameterInfo : IParameterInfo
// {
//   [propget] CentralMeridian([out] double* centralMeridian)
//   [propput] CentralMeridian([in] double centralMeridian)
// };

[
 object, uuid(5eb513c0-fe1f-11d0-ad87-080009b6f22b)
]
interface IGeographicTransform : ISpatialReferenceInfo
{
  [propget] HRESULT SourceGCS([out] IGeographicCoordinateSystem** gcs);
  [propput] HRESULT SourceGCS([in] IGeographicCoordinateSystem* gcs);
  [propget] HRESULT TargetGCS([out] IGeographicCoordinateSystem** gcs);
  [propput] HRESULT TargetGCS([in] IGeographicCoordinateSystem* gcs);
  [propget] HRESULT ParameterInfo ([out] IParameterInfo** paramInfo);
  HRESULT Forward([in] long count, [in, out, size_is(count)] WKSPoint points[]);
  HRESULT Inverse([in] long count, [in, out, size_is(count)] WKSPoint points[]);
};

[
 object, uuid(5002f420-fe1f-11d0-ad87-080009b6f22b)
]
interface IProjection : ISpatialReferenceInfo
{
   [propget] HRESULT Usage([out, retval] BSTR* usage);
   [propget] HRESULT Classification([out, retval] BSTR* classification);
   HRESULT Forward([in] long count, [in, out, size_is(count)] WKSPoint points[]);
   HRESULT Inverse([in] long count, [in, out, size_is(count)] WKSPoint points[]);
   [propget] HRESULT ParameterInfo([out, retval] IParameterInfo** paramInfo);
   [propget] HRESULT AngularUnit([out, retval] IAngularUnit** unit);
   [propput] HRESULT AngularUnit([in] IAngularUnit* unit);
   [propget] HRESULT LinearUnit([out, retval] ILinearUnit** unit);
   [propput] HRESULT LinearUnit([in] ILinearUnit* unit);
   [propget] HRESULT Ellipsoid([out, retval] IEllipsoid** ellipsoid);
   [propput] HRESULT Ellipsoid([in] IEllipsoid* ellipsoid);
};

[
 object, uuid(3dc39ff0-fe1f-11d0-ad87-080009b6f22b)
]
interface IProjectedCoordinateSystem : IGeodeticSpatialReference
{
  [propget] HRESULT Usage([out, retval] BSTR* usage);
  [propput] HRESULT Usage([in] BSTR usage);
  [propget] HRESULT GeographicCoordinateSystem
                     ([out, retval] IGeographicCoordinateSystem** gcs);
  [propput] HRESULT GeographicCoordinateSystem
                    ([in] IGeographicCoordinateSystem* gcs);
  [propget] HRESULT LinearUnit([out, retval] ILinearUnit** unit);
  [propput] HRESULT LinearUnit([in] ILinearUnit* unit);
  [propget] HRESULT Projection ([out, retval] IProjection** projection);
  [propput] HRESULT Projection ([in] IProjection* projection);
  [propget] HRESULT ParameterInfo ([out, retval] IParameterInfo** paramInfo);
  HRESULT Forward([in] long count, [in, out, size_is(count)] WKSPoint points[]);
  HRESULT Inverse([in] long count, [in, out, size_is(count)] WKSPoint points[]);
};

[
 object, uuid(620600B1-FEA1-11d0-B04B-0080C7F79481)
]
interface ISpatialReferenceFactory : IUnknown
{
  HRESULT CreateFromWKT
           ([in] BSTR wktSpatialReference, [out, retval] ISpatialReference** sref);
}

[
 object, uuid(30ae14f0-fe1f-11d0-ad87-080009b6f22b)
]
interface ISpatialReferenceAuthorityFactory : IUnknown
{
  [propget] HRESULT Authority([out] BSTR* authority);
  HRESULT CreateProjectedCoordinateSystem
           ([in] long code, [out] IProjectedCoordinateSystem** pcs);
  HRESULT CreateGeographicCoordinateSystem
           ([in] long code, [out] IGeographicCoordinateSystem** gcs);
  HRESULT CreateProjection
           ([in] long code, [out] IProjection** projection);
  HRESULT CreateGeographicTransform
           ([in] long code, [out] IGeographicTransform** gt);
  HRESULT CreateHorizontalDatum
           ([in] long code, [out] IHorizontalDatum** datum);
  HRESULT CreateEllipsoid
           ([in] long code, [out] IEllipsoid** ellipsoid);
  HRESULT CreatePrimeMeridian
           ([in] long code, [out] IPrimeMeridian** prmMerid);
  HRESULT CreateLinearUnit
           ([in] long code, [out] ILinearUnit** unit);
  HRESULT CreateAngularUnit
           ([in] long code, [out] IAngularUnit** unit);
```

```
};







#if defined (_SPATIALREFERENCE_)

// example of a specific SpatialReference library implementation

library SpatialReference
{

// COM Classes would be defined here in an actual library.
// CoTypes are shown instead.
// A CoType is a template for CoClasses.

/*************************************************************************
   coType AngularUnit
   {
      [mandatory] interface IUnit;
      [mandatory] interface IAngularUnit;
      [mandatory] interface ISpatialReferenceInfo;
   };
 coType LinearUnit
   {
      [mandatory] interface IUnit;
      [mandatory] interface ILinearUnit;
      [mandatory] interface ISpatialReferenceInfo;
   };

   coType Ellipsoid
   {
      [mandatory] interface IEllipsoid;
      [mandatory] interface ISpatialReferenceInfo;
   };

 coType HorizontalDatum
   {
      [mandatory] interface IHorizontalDatum;
      [mandatory] interface ISpatialReferenceInfo;
   };
 coType PrimeMeridian
   {
      [mandatory] interface IPrimeMeridian;
      [mandatory] interface ISpatialReferenceInfo;
   };



 coType GeographicCoordinateSystem
   {
      [mandatory] interface ISpatialReference;
      [mandatory] interface IGeodeticSpatialReference;
      [mandatory] interface IGeographicCoordinateSystem;
      [mandatory] interface ISpatialReferenceInfo;
   };

 coType Parameter
   {
      [mandatory] interface IParameter;
      [mandatory] interface ISpatialReferenceInfo;
   };

 coType ProjectedCoordinateSystem
   {
      [mandatory] interface ISpatialReference;
      [mandatory] interface IGeodeticSpatialReference;
      [mandatory] interface IProjectedCoordinateSystem;
      [mandatory] interface ISpatialReferenceInfo;
   };

// Each Projection is its own CoType, SimpleCylindrical is shown as an example

 coType SimpleCylindrical
   {
      [mandatory] interface IProjection;
      [mandatory] interface ISpatialReferenceInfo;
   };

*****************************************************************/
};
#endif
```

3.4.3 Description

3.4.3.1 ISpatialReferenceInfo Interface

The `ISpatialReferenceInfo` interface defines the standard information stored with spatial reference objects. This interface is reused for many of the spatial reference objects in the system.

3.4.3.1.1 ISpatialReferenceInfo::get_Name

```
HRESULT get_Name(BSTR* name);
```

Returns the `name` of the object.

Parameters

`name [out]`–The name of the object.

Return Code

`S_OK`—The method succeeded.

3.4.3.1.2 ISpatialReferenceInfo::put_Name

```
HRESULT put_Name(BSTR name);
```

Sets the `name` of the object.

Parameters

`name [in]`–The name of the object.

Return Code

`S_OK`—The method succeeded.

3.4.3.1.3 ISpatialReferenceInfo::get_Authority

```
HRESULT get_Authority(BSTR* authority);
```

Returns the authority name for this object, e.g., “`POSC`”, is this is a standard object with an authority specific identity code. Returns “`CUSTOM`” if this is a custom object.

Parameters

`authority[out]`–The authority for the object.

Return Code

`S_OK`—The method succeeded.

3.4.3.1.4 ISpatialReferenceInfo::put_Authority

```
HRESULT put_Authority(BSTR authority);
```

Sets the authority for the object.

Parameters

`authority [in]`–The authority for the object.

Return Code

`S_OK`—The method succeeded.

3.4.3.1.5 ISpatialReferenceInfo::get_Code

```
HRESULT get_Code(long* code);
```

Returns the authority specific identification code of the object

Parameters

`code [out]`–The authority specific identification code of the object.

Return Code

`S_OK`—The method succeeded.

3.4.3.1.6 ISpatialReferenceInfo::put_Code

```
HRESULT put_Code(long code);
```

Sets the authority specific identification code of the object.

Parameters

`code [in]`–The authority specific identification code of the object.

Return Code

`S_OK`—The method succeeded.

3.4.3.1.7 ISpatialReferenceInfo::get_Alias

```
HRESULT get_Alias(BSTR* alias);
```

Returns the alias of the object.

Parameters

`alias [out]`–The alias of the object.

Return Code

`S_OK`—The method succeeded.

3.4.3.1.8 ISpatialReferenceInfo::put_Alias

```
HRESULT put_Alias(BSTR alias);
```

Sets the alias of the object.

Parameters

`alias [in]`–The alias of the object.

Return Code

`S_OK`—The method succeeded.

3.4.3.1.9 ISpatialReferenceInfo::get_Abbreviation

```
HRESULT get_Abbreviation(BSTR* abbrev);
```

Returns the abbreviation of the object.

Parameters

`abbrev [out]`–The abbreviation of the object.

Return Code

`S_OK`—The method succeeded.

3.4.3.1.10 ISpatialReferenceInfo::put_Abbreviation

```
HRESULT put_Abbreviation(BSTR abbrev);
```

Sets the abbreviation of the object.

Parameters

`abbrev [in]`–The abbreviation of the object.

Return Code

`S_OK`—The method succeeded.

3.4.3.1.11 ISpatialReferenceInfo::get_Remarks

```
HRESULT get_Remarks(BSTR* remarks);
```

Returns the provider-supplied remarks for the object.

Parameters

`remarks [out]`–The provider-supplied remarks for the object.

Return Code

`S_OK`—The method succeeded.

3.4.3.1.12 ISpatialReferenceInfo::put_Remarks

```
HRESULT put_Remarks(BSTR remarks);
```

Sets the remarks for the object.

Parameters

`remarks [in]`–The remarks for the object.

Return Code

`S_OK`—The method succeeded.

3.4.3.1.13 ISpatialReferenceInfo::get_WellKnownText

```
HRESULT get_WellKnownText(BSTR* wkt);
```

Returns the Well-known text for this spatial reference object as defined in this proposal.

Parameters

`wkt [out]`–The Well-known text for this spatial reference object.

Return Code

`S_OK`—The method succeeded.

3.4.3.2 IUnit Interface

The `IUnit` interface abstracts different kinds of units, it has no methods.

3.4.3.3 IAngularUnit Interface

The `IAngularUnit` interface defines methods on angular units.

3.4.3.3.1 IAngularUnit::get_RadiansPerUnit

```
HRESULT get_RadiansPerUnit(double* factor);
```

Returns the number of radians per angular unit.

Parameters

`factor [out]`–The number of radians per angular unit.

Return Code

`S_OK`—The method succeeded.

3.4.3.3.2 IAngularUnit::put_RadiansPerUnit

```
HRESULT put_RadiansPerUnit(double factor);
```

Sets the number of radians per angular unit

Parameters

`factor[in]`–The number of radians per angular unit.

Return Code

`S_OK`—The method succeeded.

3.4.3.4 ILinearUnit Interface

The `ILinearUnit` interface defines methods on linear units.

3.4.3.4.1 ILinearUnit::get_MetersPerUnit

```
HRESULT get_MetersPerUnit(double* factor);
```

Returns the number of meters per linear unit.

Parameters

`factor [out]`–The number of meters per linear unit.

Return Code

`S_OK`—The method succeeded.

3.4.3.4.2 ILinearUnit::put_MetersPerUnit

```
HRESULT put_MetersPerUnit(double factor);
```

Sets the number of meters per linear unit.

Parameters

`factor[in]`–The number of meters per linear unit.

Return Code

`S_OK`—The method succeeded.

3.4.3.5 IEllipsoid Interface

The `IEllipsoid` interface defines the standard information stored with ellipsoid objects.

3.4.3.5.1 IEllipsoid::get_SemiMajorAxis

```
HRESULT get_SemiMajorAxis(double* axis);
```

Returns the value of the semi-major axis.

Parameters

`axis [out]`–The value of the semi-major axis.

Return Code

`S_OK`—The method succeeded.

3.4.3.5.2 IEllipsoid::put_SemiMajorAxis

```
HRESULT put_SemiMajorAxis(double axis);
```

Sets the value of the semi-major axis.

Parameters

`axis [in]`–The value of the semi-major axis.

Return Code

`S_OK`—The method succeeded.

3.4.3.5.3 IEllipsoid::get_SemiMinorAxis

```
HRESULT get_SemiMinorAxis(double* axis);
```

Returns the value of the semi-minor axis.

Parameters

`axis [out]`–The value of the semi-minor axis.

Return Code

`S_OK`—The method succeeded.

3.4.3.5.4 IEllipsoid::put_SemiMinorAxis

```
HRESULT put_SemiMinorAxis(double axis);
```

Sets the value of the semi-minor axis.

Parameters

`axis [in]`–The value of the semi-minor axis.

Return Code

`S_OK`—The method succeeded.

3.4.3.5.5 IEllipsoid::get_InverseFlattening

```
HRESULT get_InverseFlattening(double* invFlat);
```

Returns the value of the inverse of the flattening constant of the ellipsoid.

Parameters

invFlat [out]–The value of the inverse flattening.

Return Code

S_OK—The method succeeded.

3.4.3.5.6 IEllipsoid::put_InverseFlattening

```
HRESULT put_InverseFlattening(double invFlat);
```

Sets the value of the inverse of the flattening constant of the ellipsoid.

Parameters

invFlat [in]–The value of the inverse flattening.

Return Code

S_OK—The method succeeded.

3.4.3.5.7 IEllipsoid::get_AxisUnit

```
HRESULT get_AxisUnit(ILinearUnit** unit);
```

Returns the value of the axis unit.

Parameters

unit [out]–The value of the axis unit.

Return Code

S_OK—The method succeeded.

3.4.3.5.8 IEllipsoid::put_AxisUnit

```
HRESULT put_AxisUnit(ILinearUnit* unit);
```

Sets the value of the axis unit.

Parameters

unit [in]–The value of the axis unit

Return Code

S_OK—The method succeeded.

3.4.3.6 IHorizontalDatum Interface

The IHorizontalDatum interface defines the standard information stored with horizontal datum objects.

3.4.3.6.1 IHorizontalDatum::get_Ellipsoid

```
HRESULT get_Ellipsoid(IEllipsoid** ellipsoid);
```

Returns the ellipsoid of the datum.

Parameters

`ellipsoid [out]`–The ellipsoid of the datum.

Return Code

`S_OK`—The method succeeded.

3.4.3.6.2 IHorizontalDatum::put_Ellipsoid

```
HRESULT put_Ellipsoid(IEllipsoid* ellipsoid);
```

Sets the ellipsoid of the datum.

Parameters

`ellipsoid [in]`–The ellipsoid of the datum.

Return Code

`S_OK`—The method succeeded.

3.4.3.7 IPrimeMeridian Interface

The `IPrimeMeridian` interface defines the standard information stored with prime meridian objects. Any prime meridian object must implement this interface as well as the `ISpatialReferenceInfo` interface.

3.4.3.7.1 IPrimeMeridian::get_Longitude

```
HRESULT get_Longitude(double* longitude);
```

Returns the longitude of the prime meridian (relative to the Greenwich prime meridian).

Parameters

`longitude [out]`–The longitude of the prime meridian.

Return Code

`S_OK`—The method succeeded.

3.4.3.7.2 IPrimeMeridian::put_Longitude

```
HRESULT put_Longitude(double longitude);
```

Sets the longitude of the prime meridian (relative to the Greenwich prime meridian).

Parameters

`longitude [in]`–The longitude of the prime meridian.

Return Code

`S_OK`—The method succeeded.

3.4.3.7.3 IPrimeMeridian::get_AngularUnit

```
HRESULT get_AngularUnit(IAngularUnit** unit);
```

Returns the angular units of the prime meridian.

Parameters

`unit [out]`–The angular units of the prime meridian.

Return Code

`S_OK`—The method succeeded.

3.4.3.7.4 IPrimeMeridian::put_AngularUnit

```
HRESULT put_AngularUnit(IAngularUnit* unit);
```

Sets the angular units of the prime meridian.

Parameters

`unit [in]`–The angular units of the prime meridian.

Return Code

`S_OK`—The method succeeded.

3.4.3.8 ISpatialReference Interface

The `ISpatialReference` interface defines a root interface for all types of spatial references.

3.4.3.9 IGeodeticSpatialReference Interface

The `IGeodeticSpatialReference` interface defines a root interface for all types of geodetic spatial references, it is a subclass of `ISpatialReference`.

3.4.3.10 IGeographicCoordinateSystem Interface

The `IGeographicCoordinateSystem` interface is a subclass of `IGeodeticSpatialReference` and defines the standard information stored with geographic coordinate system objects.

3.4.3.10.1 IGeographicCoordinateSystem::get_Usage

```
HRESULT get_Usage(BSTR* usage);
```

Returns an appropriate usage comment on this Geographic Coordinate System.

Parameters

`usage [out]`–An appropriate usage comment on this geographic coordinate system.

Return Code

`S_OK`—The method succeeded.

3.4.3.10.2 IGeographicCoordinateSystem::put_Usage

```
HRESULT put_Usage(BSTR usage);
```

Sets an appropriate usage comment on this Geographic Coordinate System.

Parameters

`usage [in]`–The appropriate usage comment for this Geographic Coordinate System.

Return Code

`S_OK`—The method succeeded..

3.4.3.10.3 IGeographicCoordinateSystem::get_HorizontalDatum

```
HRESULT get_HorizontalDatum(IHorizontalDatum** datum);
```

Returns the horizontal datum of the geographic coordinate system.

Parameters

`datum [out]`–The horizontal datum of the geographic coordinate system.

Return Code

`S_OK`—The method succeeded.

3.4.3.10.4 IGeographicCoordinateSystem::put_HorizontalDatum

```
HRESULT put_HorizontalDatum(IHorizontalDatum* datum);
```

Sets the horizontal datum of the geographic coordinate system.

Parameters

`datum [in]`–The horizontal datum of the geographic coordinate system.

Return Code

`S_OK`—The method succeeded.

3.4.3.10.5 IGeographicCoordinateSystem::get_AngularUnit

```
HRESULT get_AngularUnit(IAngularUnit** unit);
```

Returns the angular units of the geographic coordinate system.

Parameters

`unit [out]`–The angular units of the geographic coordinate system.

Return Code

`S_OK`—The method succeeded.

3.4.3.10.6 IGeographicCoordinateSystem::put_AngularUnit

```
HRESULT put_AngularUnit(IAngularUnit* unit);
```

Sets the angular units of the geographic coordinate system.

Parameters

`unit [in]`–The angular units of the geographic coordinate system.

Return Code

`S_OK`—The method succeeded.

3.4.3.10.7 IGeographicCoordinateSystem::get_PrimeMeridian

```
HRESULT get_PrimeMeridian(IPrimeMeridian** prmMerid);
```

Returns the prime meridian of the geographic coordinate system.

Parameters

`prmMerid [out]`–The prime meridian of the geographic coordinate system.

Return Code

`S_OK`—The method succeeded.

3.4.3.10.8 IGeographicCoordinateSystem::put_PrimeMeridian

```
HRESULT put_PrimeMeridian(IPrimeMeridian* prmMerid);
```

Sets the prime meridian of the geographic coordinate system.

Parameters

`prmMerid [in]`–The prime meridian of the geographic coordinate system.

Return Code

S_OK—The method succeeded.

3.4.3.11 IParameter Interface

The IParameter interface is supported by parameter objects. It inherits from ISpatialReferenceInfo. A parameter is a named value.

3.4.3.11.1 IParameter::get_ValueUnit

```
HRESULT get_ValueUnit(IUnit** unit);
```

Returns the units for the parameter value.

Parameters

unit [out]–The units for the parameter value.

Return Code

S_OK—The method succeeded.

3.4.3.11.2 IParameter::put_ValueUnit

```
HRESULT put_ValueUnit(IUnit* unit);
```

Sets the units for the parameter value.

Parameters

unit [in]–The units for the parameter value.

Return Code

S_OK—The method succeeded.

3.4.3.11.3 IParameter::get_Value

```
HRESULT get_Value (double* value);
```

Returns the parameter value.

Parameters

value [out]–The parameter value..

Return Code

S_OK—The method succeeded.

3.4.3.11.4 IParameter::put_Value

```
HRESULT put_Value (double value);
```

Sets the parameter value.

Parameters

`value [in]`–The parameter value.

Return Code

`S_OK`—The method succeeded.

3.4.3.12 IParameterInfo Interface

The `IParameterInfo` interface provides an interface through which clients of a Projected Coordinate System or of a Projection can set the parameters of the projection. It provides a generic interface for discovering the names and default values of parameters, and for setting and getting parameter values. Subclasses of this interface may provide projection specific parameter access methods.

3.4.3.12.1 IParameterInfo::get_NumParameters

```
HRESULT get_NumParameters(long* numParameters);
```

Returns the number of parameters expected.

Parameters

`numParameters [out]`–The number of parameters for this projection.

Return Code

`S_OK`—The method succeeded.

3.4.3.12.2 IParameterInfo::get_DefaultParameters

```
HRESULT get_DefaultParameters(long size, IParameter* parameters[]);
```

Returns the default parameters for this projection.

Parameters

`size [in]`–The size of the parameters array passed into this function. Size should be equal to or greater than the value returned by `get_NumParameters`.

`parameters[out]`—An array of default parameters. The array must be dimensioned to hold `get_NumParameters` parameters and its size must be passed in as the first argument.

Return Code

`S_OK`—The method succeeded.

3.4.3.12.3 IParameterInfo::get_Parameters

```
HRESULT get_Parameters(long size, IParameter* parameters[]);
```

Returns the parameters set for this projection.

Parameters

`size [in]`–The size of the parameters array passed into this function. Size should be equal to or greater than the value returned by `get_NumParameters`.

`parameters[out]`—An array of parameters. The array must be dimensioned to hold `get_NumParameters` parameters and its size must be passed in as the first argument.

Return Code

`S_OK`—The method succeeded.

3.4.3.12.4 IParameterInfo::put_Parameters

```
HRESULT put_Parameters(long size, IParameter* parameters[]);
```

Set the parameters for this projection.

Parameters

`size [in]`–The size of the parameters array passed into this function. Size should be equal to or greater than the value returned by `get_NumParameters`.

`parameters[in]`—An array of parameters. The array must be dimensioned to hold `get_NumParameters` parameters and its size must be passed in as the first argument.

Return Code

`S_OK`—The method succeeded.

3.4.3.13 IGeographicTransform Interface

The `IGeographicTransform` interface is implemented on geographic transformation objects and implements datum transformations between geographic coordinate systems.

3.4.3.13.1 IGeographicTransform::get_SourceGCS

```
HRESULT get_SourceGCS(IGeographicCoordinateSystem** gcs);
```

Gets the source geographic coordinate system for the transformation.

Parameters

`gcs [out]`–The source geographic coordinate system for the transformation.

Return Code

`S_OK`—The method succeeded.

3.4.3.13.2 IGeographicTransform::put_SourceGCS

```
HRESULT put_SourceGCS(IGeographicCoordinateSystem* gcs);
```

Sets the source geographic coordinate system for the transformation.

Parameters

`gcs [in]`–The source geographic coordinate system for the transformation.

Return Code

`S_OK`—The method succeeded.

3.4.3.13.3 IGeographicTransform::get_TargetGCS

```
HRESULT get_TargetGCS(IGeographicCoordinateSystem** gcs);
```

Returns the target geographic coordinate system for the transformation.

Parameters

`gcs [out]`–The target geographic coordinate system for the transformation.

Return Code

`S_OK`—The method succeeded.

3.4.3.13.4 IGeographicTransform::put_TargetGCS

```
HRESULT put_TargetGCS(IGeographicCoordinateSystem* gcs);
```

Sets the target geographic coordinate system for the transformation.

Parameters

`gcs [in]`–The target geographic coordinate system for the transformation.

Return Code

`S_OK`—The method succeeded.

3.4.3.13.5 IGeographicTransform::Forward

```
HRESULT Forward(long count, WKSPoint* points[]);
```

Transforms an array of points from the source geographic coordinate system to the target geographic coordinate system.

Parameters

`count [in]`—The number of points to be transformed.

`points [in/out]`—On input points in the source geographic coordinate system, on output points in the target geographic coordinate system.

Return Code

S_OK—The method succeeded.

3.4.3.13.6 IGeographicTransform::Inverse

```
HRESULT Inverse(long count, WKSPoint* points[]);
```

Transforms an array of points from the target geographic coordinate system to the source geographic coordinate system.

Parameters

count [in]—The number of points to be transformed.

points [in/out]—On input points in the target geographic coordinate system, on output points in the source geographic coordinate system.

Return Code

S_OK—The method succeeded.

3.4.3.13.7 IGeographicTransform::get_ParameterInfo

```
HRESULT get_ParameterInfo(IParameterInfo** paramInfo);
```

Returns an accessor interface to the parameters for this geographic transformation.

Parameters

paramInfo [out]—An accessor interface to the parameters for this geographicc transformation.

Return Code

S_OK—The method succeeded.

3.4.3.14 IProjection Interface

The IProjection interface defines the standard information stored with projection objects. A projection object implements a coordinate transformation from a geographic coordinate system to a projected coordinate system, given the ellipsoid for the geographic coordinate system. It is expected that each coordinate transformation of interest, e.g., Transverse Mercator, Lambert, will be implemented as a COM class of coType Projection, supporting the IProjection interface.

3.4.3.14.1 IProjection::get_Usage

```
HRESULT get_Usage(BSTR* usage);
```

Returns the appropriate usage comment for the projection.

Parameters

usage [out]–The appropriate usage comment for the projection.

Return Code

`S_OK`—The method succeeded.

3.4.3.14.2 IProjection::put_Usage

```
HRESULT put_Usage(BSTR usage);
```

Sets the appropriate usage comment for the projection.

Parameters

`usage [in]`—The appropriate usage comment for the projection.

Return Code

`S_OK`—The method succeeded.

3.4.3.14.3 IProjection::get_Classification

```
HRESULT get_Classification(BSTR* classification);
```

Returns the classification comment for the projection.

Parameters

`classification [out]`–The classification comment for the projection.

Return Code

`S_OK`—The method succeeded.

3.4.3.14.4 IProjection::put_Classification

```
HRESULT put_Classification(BSTR classification);
```

Sets the classification comment for the projection.

Parameters

`classification [in]`—The classification comment for the projection.

Return Code

`S_OK`—The method succeeded.

3.4.3.14.5 IProjection::Forward

```
HRESULT Forward(long count, WKSPoint* points[]);
```

Transforms an array of points from geographic coordinates to projected coordinates.

Parameters

`count [in]`—The number of points to be projected.

`points [in/out]`—On input points in geographic (longitude, latitude) space, on output points in projected (x, y) space.

Return Code

`S_OK`—The method succeeded.

3.4.3.14.6 IProjection::Inverse

```
HRESULT Inverse(long count, WKSPoint* points[]);
```

Transforms an array of points from projected coordinates to geographic coordinates.

Parameters

`count [in]`—The number of points to be projected.

`point [in/out]`—On input a point in projected (x, y) space, on output a point in geographic (longitude, latitude) space.

Return Code

`S_OK`—The method succeeded.

3.4.3.14.7 IProjection::get_ParameterInfo

```
HRESULT get_ParameterInfo(IParameterInfo** paramInfo);
```

Returns an accessor interface to the parameters for this projection.

Parameters

`paramInfo [out]`—An accessor interface to the parameters for this projection.

Return Code

`S_OK`—The method succeeded.

3.4.3.14.8 IProjection::get_AngularUnit

```
HRESULT get_AngularUnit(IAngularUnit** unit);
```

Returns the angular units for the projection.

Parameters

`unit [out]`–The angular unit for the projection.

Return Code

`S_OK`—The method succeeded.

3.4.3.14.9 IProjection::put_AngularUnit

```
HRESULT put_AngularUnit(IAngularUnit* unit);
```

Sets the angular units for the projection.

Parameters

`unit [in]`–The angular unit for the projection.

Return Code

`S_OK`—The method succeeded.

3.4.3.14.10 IProjection::get_LinearUnit

```
HRESULT get_LinearUnit(ILinearUnit** unit);
```

Returns the linear units for the projection.

Parameters

`unit [out]`–The linear unit for the projection..

Return Code

`S_OK`—The method succeeded.

3.4.3.14.11 IProjection::put_LinearUnit

```
HRESULT put_LinearUnit(ILinearUnit* unit);
```

Sets the linear units for the projection.

Parameters

`unit [in]`–The linear unit for the projection.

Return Code

`S_OK`—The method succeeded.

3.4.3.14.12 IProjection::get_Ellipsoid

```
HRESULT get_Ellipsoid(IEllipsoid** ellipsoid);
```

Returns the ellipsoid for the projection.

Parameters

`ellipsoid [out]`–The ellipsoid for the projection.

Return Code

`S_OK`—The method succeeded.

3.4.3.14.13 IProjection::put_Ellipsoid

```
HRESULT put_Ellipsoid(IEllipsoid* ellipsoid);
```

Sets the ellipsoid for the projection.

Parameters

`ellipsoid[in]`–The ellipsoid for the projection.

Return Code

`S_OK`—The method succeeded.

3.4.3.15 IProjectedCoordinateSystem Interface

The `IProjectedCoordinateSystem` interface defines the standard information stored with projected coordinate system objects. A projected coordinate system is defined using a geographic coordinate system object and a projection object that defines the coordinate transformation from the geographic coordinate system to the projected coordinate systems. The instances of a single ProjectedCoordinateSystem COM class can be used to model different projected coordinate systems (e.g., UTM Zone 10, Albers) by associating the ProjectedCoordinateSystem instances with Projection instances belonging to different Projection COM classes (Transverse Mercator and Albers, respectively).

3.4.3.15.1 IProjectedCoordinateSystem::get_Usage

```
HRESULT get_Usage(BSTR* usage);
```

Returns the appropriate usage comment for the Projected Coordinate System.

Parameters

`usage [out]`–The appropriate usage comment for the Projected Coordinate System.

Return Code

`S_OK`—The method succeeded.

3.4.3.15.2 IProjectedCoordinateSystem::put_Usage

```
HRESULT put_Usage(BSTR usage);
```

Sets the appropriate usage comment for the Projected Coordinate System.

Parameters

`usage [in]`—The appropriate usage comment for the Projected Coordinate System.

Return Code

`S_OK`—The method succeeded.

3.4.3.15.3 IProjectedCoordinateSystem::get_GeographicCoordinateSystem

```
HRESULT get_GeographicCoordinateSystem(IGeographicCoordinateSystem** gcs);
```

Returns the geographic coordinate system associated with the projected coordinate system.

Parameters

`gcs [out]`–The geographic coordinate system associated with the projected coordinate system.

Return Code

`S_OK`—The method succeeded.

3.4.3.15.4 IProjectedCoordinateSystem::put_GeographicCoordinateSystem

```
HRESULT put_GeographicCoordinateSystem(IGeographicCoordinateSystem* gcs);
```

Sets the geographic coordinate system associated with this projected coordinate system. The projected coordinate system object sets the ellipsoid and angular units of its associated projection object based on this geographic coordinate system.

Parameters

`gcs [in]`–The geographic coordinate system associated with the projected coordinate system.

Return Code

`S_OK`—The method succeeded.

3.4.3.15.5 IProjectedCoordinateSystem::get_LinearUnit

```
HRESULT get_LinearUnit(ILinearUnit** unit);
```

Returns the linear (projected) units of the projected coordinate system.

Parameters

`unit [out]`–The linear (projected) units of the projected coordinate system.

Return Code

`S_OK`—The method succeeded.

3.4.3.15.6 IProjectedCoordinateSystem::put_LinearUnit

```
HRESULT put_LinearUnit(ILinearUnit* unit);
```

Sets the linear (projected) units for the projected coordinate system.

Parameters

`unit [in]`–The linear (projected) units for the projected coordinate system.

Return Code

S_OK—The method succeeded.

3.4.3.15.7 IProjectedCoordinateSystem::get_Projection

```
HRESULT get_Projection(IProjection** projection);
```

Returns the projection for the projected coordinate system.

Parameters

projection [out]–The projection for the projected coordinate system.

Return Code

S_OK—The method succeeded.

3.4.3.15.8 IProjectedCoordinateSystem::put_Projection

```
HRESULT put_Projection(IProjection* projection);
```

Sets the projection for the projected coordinate system.

Parameters

projection[in]–The projection for the projected coordinate system.

Return Code

S_OK—The method succeeded.

3.4.3.15.9 IProjectedCoordinateSystem::get_ParameterInfo

```
HRESULT get_ParameterInfo(IParameterInfo** paramInfo);
```

Returns an accessor interface to the parameters for this projected coordinate system, this method is forwarded on to the Projection object associated with this Projected Coordinate System.

Parameters

paramInfo [out]—An accessor interface to the parameters for this projected coordinate system.

Return Code

S_OK—The method succeeded.

3.4.3.15.10 IProjectedCoordinateSystem::Forward

```
HRESULT Forward(long count, WKSPoint* points[]);
```

Transforms an array of points from geographic coordinates to projected coordinates. This method is forwarded to the Projection object that is associated with this ProjectedCoordinateSystem for implementation.

Parameters

`count [in]`—The number of points to be projected.

`points [in/out]`—On input points in geographic (longitude, latitude) space, on output points in projected (x, y) space.

Return Code

`S_OK`—The method succeeded.

3.4.3.15.11 IProjectedCoordinateSystem::Inverse

```
HRESULT Inverse(long count, WKSPoint* points[]);
```

Transforms an array of points from projected coordinates to geographic coordinates. This method is forwarded to the Projection object that is associated with this ProjectedCoordinateSystem for implementation.

Parameters

`points [in/out]`—On input points in projected (x, y) space, on output points in geographic (longitude, latitude) space.

Return Code

`S_OK`—The method succeeded.

3.4.3.16 ISpatialReferenceFactory Interface

A Spatial Reference Factory object supports creation of spatial reference object instances given the Well-known Text representation of the instance.

3.4.3.16.1 ISpatialReferenceFactory::CreateFromWKT

```
HRESULT CreateFromWKT (BSTR wktSpatialRef, ISpatialReference** sref);
```

Creates a spatial reference object given its Well-known text representation. The output object may be either a geographic coordinate system or a projected coordinate system.

Parameters

`wktSpatialRef [in]`—The Well-known text representation for the spatial reference

`sref [out]`—The resulting spatial reference object

Return Code

`S_OK`—The method succeeded.

3.4.3.17 ISpatialReferenceAuthorityFactory Interface

A SpatialReferenceAuthorityFactory object is associated with a particular authority and creates spatial reference object instances g iven the authority specific code identifying the instance to be created.

3.4.3.17.1 ISpatialReferenceAuthorityFactory::get_Authority

```
HRESULT get_Authority(BSTR* authority);
```

Returns the authority name for this factory (e.g., “POSC”).

Parameters

authority [out]—The authority for the factory.

Return Code

S_OK—The method succeeded.

3.4.3.17.2 ISpatialReferenceAuthorityFactory::CreateProjectedCoordinateSystem

```
HRESULT CreateProjectedCoordinateSystem(long code, IProjectedCoordinateSystem**
pcs);
```

Returns a projected coordinate system object corresponding to the given code.

Parameters

code [in]—The identification code.

pcs [out]—The projected coordinate system object with the given code.

Return Code

S_OK—The method succeeded.

3.4.3.17.3 ISpatialReferenceAuthorityFactory::CreateGeographicCoordinateSystem

```
HRESULT CreateGeographicCoordinateSystem(long code,
IGeographicCoordinateSystem** gcs);
```

Returns a geographic coordinate system object corresponding to the given code.

Parameters

code [in]—The identification code.

gcs [out]—The geographic coordinate system object with the given code.

Return Code

S_OK—The method succeeded.

3.4.3.17.4 ISpatialReferenceAuthorityFactory::CreateProjection

```
HRESULT CreateProjection(long code, IProjection** projection);
```

Returns a projection object corresponding to the given code.

Parameters

`code [in]`—The identification code.

`projection [out]`—The projection object with the given code.

Return Code

`S_OK`—The method succeeded.

3.4.3.17.5 ISpatialReferenceAuthorityFactory::CreateGeographicTransform

```
HRESULT CreateGeographicTransform(long code, IGeographicTransform** gt);
```

Returns a geographic transformation object corresponding to the given code.

Parameters

`code [in]`—The identification code.

`gt [out]`—The geographic transformation object with the given code.

Return Code

`S_OK`—The method succeeded.

3.4.3.17.6 ISpatialReferenceAuthorityFactory::CreateHorizontalDatum

```
HRESULT CreateHorizontalDatum(long code, IHorizontalDatum** datum);
```

Returns a horizontal datum object corresponding to the given code.

Parameters

`code [in]`—The identification code.

`datum [out]`—The horizontal datum object with the given code.

Return Code

`S_OK`—The method succeeded.

3.4.3.17.7 ISpatialReferenceAuthorityFactory::CreateEllipsoid

```
HRESULT CreateEllipsoid(long code, IEllipsoid** ellipsoid);
```

Returns an ellipsoid object corresponding to the given code.

Parameters

`code [in]`—The identification code.

`ellipsoid [out]`—The ellipsoid object with the given code.

Return Code

`S_OK`—The method succeeded.

3.4.3.17.8 ISpatialReferenceAuthorityFactory::CreatePrimeMeridian

```
HRESULT CreatePrimeMeridian(long code, IPrimeMeridian** prmMerid);
```

Returns a prime meridian object corresponding to the given code.

Parameters

`code [in]`—The identification code.

`prmMerid [out]`—The prime meridian object with the given code.

Return Code

`S_OK`—The method succeeded.

3.4.3.17.9 ISpatialReferenceAuthorityFactory::CreateLinearUnit

```
HRESULT CreateLinearUnit(long code, ILinearUnit** unit);
```

Returns a linear unit object corresponding to the given code.

Parameters

`code [in]`—The identification code.

`unit [out]`—The linear unit object with the given code.

Return Code

`S_OK`—The method succeeded.

3.4.3.17.10 ISpatialReferenceAuthorityFactory::CreateAngularUnit

```
HRESULT CreateAngularUnit(long code, IAngularUnit** unit);
```

Returns an angular unit object corresponding to the given code.

Parameters

`code [in]`—The identification code.

`unit [out]`—The angular unit object with the given code.

Return Code

S_OK—The method succeeded.

3.4.4 Exceptions, Errors, and Error Codes

All of the error codes for spatial reference system objects are returned in the HRESULTs of the methods of the interfaces. Any return value not equal to S_OK will be interpreted as a provider specific error. These objects do not throw exceptions.

3.5 Well-known Text Representation of Spatial Reference Systems

3.5.1 Component Overview

The Well-known Text Representation of Spatial Reference Systems provides a standard textual representation for spatial reference system information.

3.5.2 Component Description

The definitions of the well-known text representation are modeled after the POSC/EPSG coordinate system data model.

A spatial reference system, also referred to as a coordinate system, is a geographic (latitude-longitude), a projected (X,Y), or a geocentric (X,Y,Z) coordinate system.

The coordinate system is composed of several objects. Each object has a keyword in upper case (for example, DATUM or UNIT) followed by the defining, comma-delimited, parameters of the object in brackets. Some objects are composed of objects so the result is a nested structure. Implementations are free to substitute standard brackets ( ) for square brackets [ ] and should be prepared to read both forms of brackets.

The EBNF (Extended Backus Naur Form) definition for the string representation of a coordinate system is as follows, using square brackets, see note above:

```
<coordinate system> = <projected cs> | <geographic cs> | <geocentric cs>

<projected cs> = PROJCS["<name>", <geographic cs>, <projection>, {<parameter>,}* <linear unit>]

<projection> = PROJECTION["<name>"]

<parameter> = PARAMETER["<name>", <value>]

<value> = <number>
```

A data set's coordinate system is identified by the PROJCS keyword if the data are in projected coordinates, by GEOGCS if in geographic coordinates, or by GEOCCS if in geocentric coordinates.

The PROJCS keyword is followed by all of the "pieces" which define the projected coordinate system. The first piece of any object is always the name. Several objects follow the projected coordinate system name: the geographic coordinate system, the map projection, 1 or more parameters, and the linear unit of measure. All projected coordinate systems are based upon a geographic coordinate system so we will describe the pieces specific to a projected coordinate system first. As an example, UTM zone 10N on the NAD83 datum is defined as:

```
            PROJCS["NAD_1983_UTM_Zone_10N",
                <geographic cs>,
                PROJECTION["Transverse_Mercator"],
                PARAMETER["False_Easting",500000.0],
                PARAMETER["False_Northing",0.0],
                PARAMETER["Central_Meridian",-123.0],
                PARAMETER["Scale_Factor",0.9996],
                PARAMETER["Latitude_of_Origin",0.0],
                UNIT["Meter",1.0]]
```

The name and several objects define the geographic coordinate system object in turn: the datum, the prime meridian, and the angular unit of measure.

```
<geographic cs> = GEOGCS["<name>", <datum>, <prime meridian>, <angular unit>]

<datum> = DATUM["<name>", <spheroid>]

<spheroid> = SPHEROID["<name>", <semi-major axis>, <inverse flattening>]

<semi-major axis> = <number>   NOTE: semi-major axis is measured in meters and must be > 0.

<inverse flattening> = <number>

<prime meridian> = PRIMEM["<name>", <longitude>]

<longitude> = <number>
```

The geographic coordinate system string for UTM zone 10 on NAD83 is

```
           GEOGCS["GCS_North_American_1983",
               DATUM["D_North_American_1983",
               SPHEROID["GRS_1980",6378137,298.257222101]],
               PRIMEM["Greenwich",0],
               UNIT["Degree",0.0174532925199433]]
```

The `UNIT` object can represent angular or linear unit of measures.

```
<angular unit> = <unit>

<linear unit> = <unit>

<unit> = UNIT["<name>", <conversion factor>]

<conversion factor> = <number>
```

`<conversion factor>` specifies number of meters (for a linear unit) or number of radians (for an angular unit) per unit and must be greater than zero.

So the full string representation of UTM Zone 10N is

```
PROJCS["NAD_1983_UTM_Zone_10N",
           GEOGCS["GCS_North_American_1983",
           DATUM[  "D_North_American_1983",SPHEROID["GRS_1980",6378137,298.257222101]],
           PRIMEM["Greenwich",0],UNIT["Degree",0.0174532925199433]],
           PROJECTION["Transverse_Mercator"],PARAMETER["False_Easting",500000.0],
           PARAMETER["False_Northing",0.0],PARAMETER["Central_Meridian",-123.0],
           PARAMETER["Scale_Factor",0.9996],PARAMETER["Latitude_of_Origin",0.0],
           UNIT["Meter",1.0]]
```

A geocentric coordinate system is quite similar to a geographic coordinate system. It is represented by

```
<geocentric cs> = GEOCCS["<name>", <datum>, <prime meridian>, <linear unit>]
```

4 Supported Spatial Reference Data

4.1 Supported Linear Units

Meter	1.0
Foot (International)	0.3048
U.S. Foot	12/39.37
Modified American Foot	12.0004584/39.37
Clarke's Foot	12/39.370432
Indian Foot	12/39.370141
Link	7.92/39.370432
Link (Benoit)	7.92/39.370113
Link (Sears)	7.92/39.370147
Chain (Benoit)	792/39.370113
Chain (Sears)	792/39.370147
Yard (Indian)	36/39.370141
Yard (Sears)	36/39.370147
Fathom	1.8288
Nautical Mile	1852.0

4.2 Supported Angular Units

Radian	1.0
Decimal Degree	π/180
Decimal Minute	(π/180)/60
Decimal Second	(π/180)/36000
Gon	π/200
Grad	π/200

4.3 Supported Spheroids

Name	**Semi-major Axis**	**Inverse Flattening**
Airy	6377563.396	299.3249646
Modified Airy	6377340.189	299.3249646
Australian	6378160	298.25
Bessel	6377397.155	299.1528128
Modified Bessel	6377492.018	299.1528128
Bessel (Namibia)	6377483.865	299.1528128
Clarke 1866	6378206.4	294.9786982
Clarke 1866 (Michigan)	6378693.704	294.978684677
Clarke 1880	6378249.145	293.465
Clarke 1880 (Arc)	6378249.145	293.466307656

Clarke 1880 (Benoit)	6378300.79	293.466234571
Clarke 1880 (IGN)	6378249.2	293.46602
Clarke 1880 (RGS)	6378249.145	293.465
Clarke 1880 (SGA)	6378249.2	293.46598
Everest 1830	6377276.345	300.8017
Everest 1975	6377301.243	300.8017
Everest (Sarawak and Sabah)	6377298.556	300.8017
Modified Everest 1948	6377304.063	300.8017
Fischer 1960	6378166	298.3
Fischer 1968	6378150	298.3
Modified Fischer (1960)	6378155	298.3
GEM10C	6378137	298.257222101
GRS 1980	6378137	298.257222101
Hayford 1909	6378388	297.0
Helmert 1906	6378200	298.3
Hough	6378270	297.0
International 1909	6378388	297.0
International 1924	6378388	297.0
New International 1967	6378157.5	298.2496
Krasovsky	6378245	298.3
Mercury 1960	6378166	298.3
Modified Mercury 1968	6378150	298.3
NWL9D	6378145	298.25
OSU_86F	6378136.2	298.25722
OSU_91A	6378136.3	298.25722
Plessis 1817	6376523	308.64
South American 1969	6378160	298.25
Southeast Asia	6378155	298.3
Sphere (radius = 1.0)	1	0
Sphere (radius = 6371000 m)	6371000	0
Sphere (radius = 6370997 m)	6370997	0
Struve 1860	6378297	294.73
Walbeck	6376896	302.78
War Office	6378300.583	296
WGS 1960	6378165	298.3
WGS 1966	6378145	298.25
WGS 1972	6378135	298.26
WGS 1984	6378137	298.257223563

4.4 Supported Geodetic Datums

Adindan
Afgooye
Agadez
Australian Geodetic Datum 1966
Australian Geodetic Datum 1984
Ain el Abd 1970
Amersfoort
Aratu
Arc 1950
Arc 1960
Ancienne Triangulation Francaise
Barbados
Batavia
Beduaram
Beijing 1954
Reseau National Belge 1950
Reseau National Belge 1972
Bermuda 1957
Bern 1898
Bern 1938

Lisbon
Loma Quintana
Lome
Luzon 1911
Mahe 1971
Makassar
Malongo 1987
Manoca
Massawa
Merchich
Militar-Geographische Institute
Mhast
Minna
Monte Mario
M'poraloko
NAD Michigan
North American Datum 1927
North American Datum 1983
Nahrwan 1967
Naparima 1972

Bogota
Bukit Rimpah
Camacupa
Campo Inchauspe
Cape
Carthage
Chua
Conakry 1905
Corrego Alegre
Cote d'Ivoire
Datum 73
Deir ez Zor
Deutsche Hauptdreiecksnetz
Douala
European Datum 1950
European Datum 1987
Egypt 1907
European Reference System 1989
Fahud
Gandajika 1970
Garoua
Geocentric Datum of Australia 1994
Guyane Francaise
Herat North
Hito XVIII 1963
Hu Tzu Shan
Hungarian Datum 1972
Indian 1954
Indian 1975
Indonesian Datum 1974
Jamaica 1875
Jamaica 1969
Kalianpur
Kandawala
Kertau
Kuwait Oil Company
La Canoa
Lake
Leigon
Liberia 1964
Nord de Guerre
NGO 1948
Nord Sahara 1959
NSWC 9Z-2
Nouvelle Triangulation Francaise
New Zealand Geodetic Datum 1949
OS (SN) 1980
OSGB 1936
OSGB 1970 (SN)
Padang 1884
Palestine 1923
Pointe Noire
Provisional South American Datum 1956
Pulkovo 1942
Qatar
Qatar 1948
Qornoq
RT38
South American Datum 1969
Sapper Hill 1943
Schwarzeck
Segora
Serindung
Stockholm 1938
Sudan
Tananarive 1925
Timbalai 1948
TM65
TM75
Tokyo
Trinidad 1903
Trucial Coast 1948
Voirol 1875
Voirol Unifie 1960
WGS 1972
WGS 1972 Transit Broadcast Ephemeris
WGS 1984
Yacare
Yoff
Zanderij

4.5 Supported Prime Meridians

Greenwich	0° 0' 0"
Bern	7° 26' 22.5" E
Bogota	74° 4' 51.3" W
Brussels	4° 22' 4.71" E
Ferro	17° 40' 0" W
Jakarta	106° 48' 27.79" E
Lisbon	9° 7' 54.862" W
Madrid	3° 41' 16.58" W
Paris	2° 20' 14.025"E
Rome	12° 27' 8.4" E
Stockholm	18° 3' 29" E

4.6 Supported Map Projections

Cylindrical Projections
Behrmann
Cassini
Cylindrical equal area

Pseudocylindrical Projections
Craster parabolic
Eckert I
Eckert II

Equirectangular
Gall's stereographic
Gauss-Kruger
Mercator
Miller cylindrical
Oblique Mercator (Hotine)
Plate-Carée
Times
Transverse Mercator

Eckert III
Eckert IV
Eckert V
Eckert VI
McBryde-Thomas flat polar quartic
Mollweide
Robinson
Sinusoidal (Sansom-Flamsteed)
Winkel I

Conic Projections
Albers conic equal-area
Bipolar oblique conformal conic
Bonne
Equidistant conic
Lambert conformal conic
Polyconic
Simple conic

Modified
Chamberlin trimetric
Two-point equidistant
Hammer-Aitoff equal-area

Miscellaneous
Alaska series E
Alaska Grid (Modified-Stereographic by Snyder)
Van der Grinten I

Azimuthal or Planar Projections
Azimuthal equidistant
General vertical near-side perspective
Gnomonic
Lambert Azimuthal equal-area
Orthographic
Polar Stereographic
Stereographic

4.7 Map Projection Parameters

central_meridian	the line of longitude chosen as the origin of x-coordinates.
scale_factor	used generally to reduce the amount of distortion in a map projection.
standard_parallel_1	a line of latitude that has no distortion generally. Also used for "latitude of true scale."
standard_parallel_2	a line of latitude that has no distortion generally.
longitude_of_center	the longitude which defines the center point of the map projection.
latitude_of_center	the latitude which defines the center point of the map projection.
latitude_of_origin	the latitude chosen as the origin of y-coordinates.
false_easting	added to x-coordinates. Used to give positive values.
false_northing	added to y-coordinates. Used to give positive values.
azimuth	the angle east of north which defines the center line of an oblique projection.
longitude_of_point_1	the longitude of the first point needed for a map projection.
latitude_of_point_1	the latitude of the first point needed for a map projection.
longitude_of_point_2	the longitude of the second point needed for a map projection.
latitude_of_point_2	the latitude of the second point needed for a map projection.
longitude_of_point_3	the longitude of the third point needed for a map projection.
latitude_of_point_3	the latitude of the third point needed for a map projection.
landsat_number	the number of a Landsat satellite.
path_number	the orbital path number for a particular satellite.
perspective_point_height	the height above the earth of the perspective point of the map projection.
fipszone	State Plane Coordinate System zone number.
zone	UTM zone number.

5 References

1. The OpenGIS Abstract Specification: An Object Model for Interoperable Geoprocessing, Revision 1, OpenGIS Consortium, Inc, OpenGIS Project Document Number 96-015R1, 1996.

2. OpenGIS Project Document 96-025: Geodetic Reference Systems, OpenGIS Consortium, Inc, October 14, 1996.

3. POSC (Petrotechnical Open Software Consortium) Epicentre Model V2.1, fttp://posc.org/public/geodetic, July 1995.

4. Clementini, Eliseo, Di Felice, P., van Oostrom, p., A Small Set of Formal Topological Relationships Suitable for End-User Interaction, in D. Abel and B. C. Ooi (Ed.), Advances in Spatial Databases—Third International Symposium. SSD '93. LNCS 692. Pp. 277-295. Springer-Verlag. Singapore (1993).

5. Clementini E. and Di Felice P., A Comparison of Methods for Representing Topological Relationships, Information Sciences 80, 1-34, 1994.

6. Clementini, Eliseo, Di Felice, P., A Model for Representing Topological Relationships Between Complex Geometric Features in Spatial Databases, Information Sciences 90 (1-4):121-136 , 1996.

7. Clementini E., Di Felice P and Califano, G. Composite Regions in Topological Queries, Information Systems, v 20, no 6, pp 33-48, 1995.

8. Egenhofer, M.F. and Franzosa, Point Set Topological Spatial Relations, International Journal of Geographical Information Systems, vol 5, no 2, 161-174, 1991.

9. Egenhofer, M.J., Clementini, E. and Di Felice, P., Topological relations between regions with holes, International Journal of Geographical Information Systems, vol 8, no 2, pp 129—142, 1994.

10. Egenhofer, M.J. and Herring, J., A mathematical framework for the definition of topological relationships. Proceedings of the Fourth International Symposium on Spatial Data Handling, Columbus, Ohi, pp. 803-813.

11. Egenhofer M.J. and Herring, J., Categorizing binary topological relationships between regions, lines and points in geographic databases, Tech. Report., Department of Surveying Engineering, University of Maine, Orono, ME 1991.

12. Egenhofer. M.J. and Sharma, J., Topological Relations between regions in $\Re^2$ and Z^2, Advances in Spatial Databases—Third International Symposium, SSD ’93, vol. 692, Lecture Notes in Computer Science, pp. 36-52, Springer Verlag, Singapore (1993).

13. Worboys, M.F. and Bofakos, P. A Canonical model for a class of areal spatial objects, Advances in Spatial Databases—Third International Symposium, SSD ’93, vol. 692, Lecture Notes in Computer Science, pp. 36-52, Springer Verlag, Singapore (1993).

14. Worboys, M.F. A generic model for planar geographical objects, International Journal of Geographical Information Systems, 1992, vol 6, no 5, 353-372.

15. http://www.omg.org/corba/sectrans.htm : CORBAservices : Common Object Services Specification, Ch 8. Externalization Service Specification, OMG.

16. http://www.microsoft.com/oledev : Distributed Component Object Model Protocol Specification—DCOM 1.0, Microsoft Corporation.